FACULTAD DE INGENIERÍA
UNIVERSIDAD DE LA REPÚBLICA

Especificación de Contratos de Software con JML Y Eiffel

Proyecto de Grado

Vanessa Casella
vanecasella22@gmail.com

Gabriel Arrospide
garrospide@gmail.com

Tutores

Silvana Moreno
smoreno@fing.edu.uy

Diego Vallespir
dvallesp@fing.edu.uy

Montevideo, Uruguay
Marzo, 2014

Resumen

En el presente trabajo se introduce el enfoque de diseño por contratos que se basa en los métodos formales para el diseño e implementación de aplicaciones y componentes. Se realiza un análisis de dos lenguajes de especificación formal, JML y Eiffel, que siguen el paradigma de diseño por contrato. JML permite realizar la especificación de productos de software desarrollados en Java y Eiffel es un lenguaje de programación pionero en introducir la noción de diseño por contratos.

Se realiza un estudio de las herramientas vinculadas al JML, que permiten interpretar y ejecutar las especificaciones realizadas en dicho lenguaje. Se propone una nueva caracterización a partir del análisis de dos categorizaciones de herramientas, propuestas en otros trabajos.

En el estudio de los dos lenguajes de especificación formal se realizan ejercicios prácticos, en particular se propuso la implementación y especificación de una Lista dinámica, la cual involucra la manipulación de punteros, comportamiento que nos pareció interesante de especificar en ambos lenguajes. Para la resolución del ejercicio en el JML se utilizaron algunas de las herramientas estudiadas. Los principales factores utilizados para elegir las mismas fueron de acuerdo a la documentación disponible y continuidad de versiones.

Índice general

Resumen III

1. Introducción 1
- 1.1. Diseño por Contratos . . . 2
- 1.2. Objetivos . . . 3
- 1.3. Trabajo Realizado . . . 4
- 1.4. Estructura del Documento . . . 4

2. Java Modeling Language 5
- 2.1. Especificación de la interfaz de comportamiento . . . 5
  - 2.1.1. Especificación de métodos . . . 6
  - 2.1.2. Especificaciones de varios comportamientos . . . 6
  - 2.1.3. Especificación de ejemplos . . . 8
  - 2.1.4. Especificaciones de peso ligero y peso pesado . . . 8
  - 2.1.5. Especificaciones informales . . . 9
- 2.2. Formalización del lenguaje . . . 9
  - 2.2.1. Expresión de especificación . . . 9
  - 2.2.2. Modificadores . . . 13
  - 2.2.3. Tipo de Especificaciones . . . 16
  - 2.2.4. Grupos de datos . . . 18
  - 2.2.5. Herencia . . . 19
- 2.3. Aplicación a problemas de algoritmia . . . 20
- 2.4. Restricciones del lenguaje . . . 20
- 2.5. Estado y planes para el JML . . . 20

3. Herramientas 23
- 3.1. Categorizaciones existentes de herramientas . . . 23
- 3.2. Nueva Categorización . . . 24
  - 3.2.1. Compilación de clases Java anotadas con el JML. . . . 25
  - 3.2.2. Chequeo en tiempo de ejecución de *assertions* . . . 25
  - 3.2.3. Generación y ejecución de casos de pruebas . . . 26
  - 3.2.4. Chequeo estático y verificación . . . 26
  - 3.2.5. Generación de especificaciones . . . 28
  - 3.2.6. Documentación . . . 29

4. Eiffel **31**
4.1. El lenguaje . . . 31
4.2. Formalización del Lenguaje . . . 31
4.2.1. Implementación de Clases . . . 31
4.2.2. Características . . . 34
4.2.3. Herencia . . . 34
4.2.4. Entidades especiales de Eiffel . . . 36
4.2.5. Conformidad . . . 36
4.2.6. Convertibilidad . . . 37
4.2.7. Herencia Repetida . . . 37
4.2.8. Estructuras de Control . . . 37
4.2.9. Creación de Objetos . . . 38
4.2.10. Comparación y copia de tipos . . . 39
4.2.11. Llamada a características . . . 40
4.2.12. Agentes . . . 40
4.2.13. Manejo de excepciones . . . 40
4.2.14. Pre-condiciones y Post-condiciones . . . 40

5. De la teoría a la práctica **47**
5.1. Especificación en JML . . . 49
5.2. Especificación en Eiffel . . . 57

6. Conclusiones y Trabajos a Futuro **63**
6.1. Conclusiones . . . 63
6.2. Trabajos a Futuro . . . 64

Índice de figuras

2.1. Cláusula Refine . . . 5
2.2. Especificación de pre y post condición . . . 6
2.3. Especificación excepcional . . . 6
2.4. Especificaciones normales de varios comportamientos . . . 7
2.5. Especificaciones normales y excepcionales de varios comportamientos . . . 8
2.6. Especificación de ejemplo . . . 8
2.7. Especificaciones informales . . . 9
2.8. Ejemplo de uso de Operadores Lógicos . . . 10
2.9. Cuantificadores . . . 11
2.10. Operadores del JML . . . 13
2.11. Ejemplo de uso del Modificador *Puro* . . . 15
2.12. Modificador *Modelo* . . . 15
2.13. Modificador *Fantasma* . . . 16
2.14. Invariante de bucle . . . 17
2.15. Invariante de clase . . . 17
2.16. *History Constraints* . . . 18
2.17. *Cláusula assignable* . . . 18
2.18. *Grupo de datos* . . . 19
2.19. Herencia . . . 20

4.1. Clase Derivada . . . 32
4.2. Clase Congelada . . . 33
4.3. Clase Expadida . . . 34
4.4. Características . . . 34
4.5. Herencia Multiple . . . 35
4.6. Herencia . . . 36
4.7. Creación de Objetos . . . 39
4.8. Pre-condiciones y Post-condiciones . . . 42
4.9. Pre-condiciones y Post-condiciones con iteración . . . 43
4.10. Invariantes . . . 44
4.11. Cláusula OLD . . . 45

5.1. Diseño de Lista . . . 47
5.2. Especificación del método *insertar* utilizando la función *get* en JML . . . 52
5.3. Especificación del método *insertar* utilizando la cláusula *reach* en JML . . . 54
5.4. Especificación del método *insertar* utilizando *modelos* en JML . . . 56
5.5. Especificación e implementación del método *insertar* en Eiffel . . . 61

Índice de cuadros

1.1. Contratos de software 2

2.1. Descripción de algunos operadores lógicos 10
2.2. Descripción de algunos Cuantificadores 10
2.3. Descripción de algunos operadores del JML 13

Capítulo 1

Introducción

La Ingeniería de Software es la aplicación de un enfoque sistemático, disciplinado y cuantificable al desarrollo, operación y mantenimiento de software, es decir, la aplicación de la ingeniería al software [1]. Hoy en día resulta crítico el desarrollo de productos de software con la menor cantidad de defectos posibles y preocupa la dificultad para diseñar y desarrollar productos de software *cero defectos*. Por ello es necesario la aplicación de enfoques y metodologías que permitan un desarrollo de software de calidad de manera eficiente. El uso de los métodos formales aporta grandes ventajas en la Ingeniería de Software, mejorando la calidad de los programas.

Los métodos formales usados en el desarrollo de sistemas de computadoras son técnicas, basadas en la matemática, que describen propiedades del sistema. Estos métodos proveen un entorno de trabajo donde las personas pueden especificar, desarrollar y verificar los sistemas de manera sistemática [2]. El uso de notaciones y lenguajes formales permite plantear los requerimientos de un sistema sin ambigüedades y lograr que los mismos sean confiables y seguros. A causa de ello se desarrollan nuevos lenguajes y herramientas para especificar y modelar formalmente sistemas, así como también el diseño de metodologías para verificarlos y validarlos. Los métodos formales son utilizados en la verificación formal, que consiste en probar o refutar la correctitud de un programa con respecto a la especificación formal.

En el marco de este proyecto de grado se estudia el enfoque de diseño por contratos (DBC) que se basa en los métodos formales para el diseño e implementación de aplicaciones y componentes. Se establece un contrato que debe cumplirse entre elementos de software que brindan funcionalidades, llamados proveedores, y quienes consumen dichas funcionalidades, llamados clientes. La utilización de DBC favorece la construcción de productos de software, no solo desde el punto de vista de la verificación, sino también porque permite mantener control sobre el comportamientos de rutinas y clases que se quiera especificar. Esto le da valor agregado al proyecto de software y genera confianza tanto en implementadores como en clientes. En la siguiente sección se introducirá el concepto de diseño por contratos.

El desarrollo de software de calidad implica la utilización de metodologías o procedimientos estándares para el análisis, diseño, programación y prueba del software para lograr una mayor confiabilidad. En este contexto, se estudian lenguajes de especificación formal, que siguen el paradigma de diseño por contrato, y herramientas de verificación que proveen funcionalidades basadas en dichos lenguajes. Actualmente los diversos cursos obligatorios dictados en la carrera de grado Ingeniería en Computación, no incluyen el aprendizaje de métodos formales y diseño por contrato, por lo que este proyecto busca estudiar lenguajes de especificación formal que utilizan dicho enfoque. Se estudia JML dado que es el lenguaje de especificación más conocido para Java, y Eiffel por ser el lenguaje pionero en este tipo de especificación. Además se desea estudiar las diversas herramientas disponibles para dichos lenguajes.

1.1. Diseño por Contratos

El diseño por contrato es una metodología utilizada en el desarrollo de software, que se basa en los métodos formales. En DBC se establece un contrato entre una operación o subprograma (proveedor) y quien lo invoca (cliente). El cliente debe garantizar ciertas condiciones antes de llamar al proveedor, denominada **pre-condición**, en cambio el proveedor garantiza ciertas condiciones que se cumplirán después de dicha invocación, llamada **post-condición**. Un contrato de software especifica las obligaciones y derechos de proveedores y clientes.

El cuerpo de una rutina invocada no debe chequear el cumplimiento de la pre-condición, ya que la misma define las condiciones por las cuales una llamada a la rutina es válida. Por otra parte, las post-condiciones establecen las obligaciones que la rutina debe cumplir al finalizar su ejecución. Los contratos de software se especifican utilizando lenguajes formales. Un lenguaje de especificación formal provee una notación (su dominio sintáctico), un universo de objetos (su dominio semántico), y una regla precisa que define qué objetos satisfacen cada especificación. Una especificación es una sentencia escrita en términos de los elementos de su dominio sintáctico, que denota un conjunto de especificandos. Un especificando es un objeto que satisface una especificación, es un subconjunto del dominio semántico. La relación de satisfacción provee el significado o interpretación de los elementos sintácticos [2]. En el Cuadro 1.1 se describen los contratos de software.

Rol	**Obligaciones**	**Beneficios**
Proveedor	Satisfacer la post-condición	No debe preocuparse por las condiciones de la pre-condición
Cliente	Satisfacer la pre-condición	Obtiene el resultado establecido en la post-condición

Cuadro 1.1: Contratos de software

Como ejemplo se define una clase en Java, llamada *Persona*, con atributos *nombre*, *edad* y *peso* y los métodos *setters* y funciones *getters* de dichos atributos. A modo de ejemplo se considera solo el método *setEdad*, que setea el atributo edad a partir de un parámetro de entrada. A continuación se presenta la clase *Persona* y las pre-condiciones y post-condiciones de manera informal del método *setEdad*.

```
public class Persona {
   private String nombre;
   private double peso;
   private int edad;

   ...

   Pre-condición:    El parámetro de entrada tiene que ser
                     mayor o igual a 0.
   Post-condición:   El atributo 'edad' tiene el mismo valor
                     que el parámetro de entrada.
   public void setEdad(int valor) {
      edad = valor;
   }
}
```

En la implementación del método no se chequea que el parámetro de entrada sea mayor o igual a 0, ya que queda explícito en la pre-condición.

Se define otra clase llamada *Familia*, la cual esta compuesta por el atributo *integrantes*, que contiene los integrantes de la familia. Dado el método *agregarNuevoIntegrante*, se definen las siguientes pre-condiciones y post -condiciones.

```
public class Familia {
   private List<Persona> integrantes;
   ...

   Pre-condición:    El parámetro de entrada 'nombre' debe
                     ser distinto de vacio.
   Pre-condición:    El parámetro de entrada 'peso' debe ser
                     mayor o igual a cero.
   Post-condición:   Se agrega al final de la lista
                     'integrantes' una instancia de la clase
                     'Persona' con atributos 'nombre' y
                     'peso' igual a los parámetros recibidos,
                     y se setea en 0 el atributo edad.

   public void agregarNuevoIntegrante
               (String nombre, double peso) {
      Persona p = new Persona();
      p.setEdad(0);
      p.setNombre(nombre);
      p.setPeso(peso);
      integrantes.add(p);
   }
}
```

En el contexto de diseño por contrato, la clase *Familia* tiene el rol de **cliente** al invocar la función *setEdad*, definida en la clase *Persona*, desde el método *agregarNuevoIntegrante*. A su vez la clase *Persona* tiene el rol de **proveedor** al brindar esta funcionalidad. El cliente (clase *Familia*) asume que el proveedor (clase *Persona*) cumple con el contrato de dicha función, así como también el proveedor asume que el parámetro de entrada *edad* será presentado de forma correcta por el cliente.

1.2. Objetivos

Existen cinco objetivos principales a cumplir en este proyecto. En primer lugar comprender y familiarizarse con la metodología de diseño por contratos.

El segundo objetivo consiste en estudiar en profundidad el lenguaje de especificación formal JML, para realizar la especificación de productos de software desarrollados en Java. Se desea comprender su sintaxis y semántica, así como también especificar algunos ejercicios prácticos.

Como tercer objetivo se desea conocer las herramientas vinculadas al JML, las cuales permiten interpretar y ejecutar las especificaciones realizadas, además de dar soporte al momento de diseñar e implementar las mismas.

El cuarto objetivo consiste en estudiar el lenguaje Eiffel como otra alternativa de lenguaje formal de especificación. Se busca al igual que con el JML, especificar formalmente elementos de software, utilizando Eiffel como lenguaje de especificación formal.

El último de los objetivos consiste en especificar un ejercicio práctico en ambos lenguajes y probar el mismo en algunas de las herramientas estudiadas.

1.3. Trabajo Realizado

Para cumplir con los objetivos se realizó un estudio en profundidad del JML, tomando como fuente de estudio diversas publicaciones de la página oficial del lenguaje, muchas de las cuales tenían la participación de *Gary T. Leavens*, responsable del proyecto. Se estudiaron y pusieron en práctica diferentes ejercicios y ejemplos presentados por estudiantes de diversas universidades. Además fue necesario profundizar conceptos, implementando y especificando nuevas clases y rutinas que contengan comportamientos no contemplados por los ejercicios estudiados.

Se investigaron diversas herramientas distribuidas en dos categorizaciones propuestas en otros trabajos [12, 13]. Dado que las categorizaciones anteriores no contemplaban todas las herramientas existentes y algunas categorías, a nuestro criterio, necesitaban ser divididas, se propuso una nueva categorización a partir de ellas. Se examinaron en detalle algunas herramientas pertenecientes a la categoría *Chequeo estático y verificación*, las cuales generan pruebas formales basadas en la especificación realizada. También se consideró la categoría *Chequeo en tiempo de ejecución de assertions*, que hace referencia a herramientas utilizadas para la ejecución de clases Java compiladas con un compilador de JML. Además se estudiaron las herramientas dedicadas a la compilación de código Java anotado con especificaciones en JML, perteneciente a la categoría *Compilación de clases Java anotadas con JML*. Otra categoría que se consideró fue *Generación y ejecución de casos de prueba*, las herramientas que la incluyen pueden generar de forma automática casos de pruebas. Por último también se examinaron herramientas que permiten documentar el código anotado con JML, las mismas pertenecen a la categoría *Documentación*.

Se estudió el lenguaje de especificación formal Eiffel como lenguaje para el desarrollo y especificación de productos de software, dado que es el pionero en la especificación de elementos de este tipo basados en la metodología de diseño por contrato. Dicho estudio se realizó utilizando los tutoriales brindados por la página oficial del lenguaje y cursos dictados por *Hal Webre*, capacitador oficial del lenguaje, y de la publicación de ECMA - 367 *Eiffel:Analysis, Design and Programming Language* del año 2006. Se realizaron diversas pruebas, al igual que con JML, para poder comprender en profundidad la semántica y así especificar ejemplos complejos. Se estudió además el entorno oficial de desarrollo del lenguaje, el cual proporciona en un solo producto de software las herramientas necesarias para la implementación, especificación, compilación y depuración de los elementos de software.

Luego de examinar ambos lenguajes resultó interesante llevar a la práctica los conceptos adquiridos. Se buscó realizar diversos ejercicios prácticos con dichos lenguajes, algunos de ellos presentados en los capítulos *Java Modeling Language* y *Eiffel*, y en particular se propuso la implementación y especificación de la clase *Lista*, la cual involucra la manipulación de punteros, comportamiento que nos pareció interesante de especificar con ambos lenguajes. Este ejercicio se presenta en uno de los capítulos del documento. Esta especificación formó parte de los ejercicios prácticos utilizados para estudiar el estado actual de algunas herramientas.

1.4. Estructura del Documento

El presente documento se estructura en 5 capítulos además del actual. El capítulo **Java Modeling Language** describe las principales características de el JML. Se presenta su sintaxis y semántica. Además se muestran restricciones y aplicaciones de dicho lenguaje.

En el capítulo **Herramientas** se presentan herramientas que implementan el JML categorizadas por diferentes autores. Se realiza una descripción de cada una de ellas y se crea una nueva categorización.

El capítulo **Eiffel** describe las características principales del lenguaje Eiffel, detallando su sintaxis y semántica.

En el capítulo **De la teoría a la práctica** se aplican los conceptos adquiridos en los capítulos anteriores para resolver un problema específico utilizando el JML y Eiffel como lenguajes formales, y así evaluar si es posible su especificación en ambos lenguajes. Además se examina el estado actual de algunas herramientas utilizadas.

Por último en el capítulo **Conclusiones y Trabajos a Futuro** se presentan las conclusiones sobre este trabajo. Además se describen los trabajos que consideramos interesantes para abordar y desarrollar en un futuro.

Capítulo 2

Java Modeling Language

Java Modeling Language (JML) es un lenguaje formal de especificación para el diseño por contrato (DBC), que define el comportamiento de una interfaz para programas escritos en Java. Combina el sentido práctico de Eiffel [3] con la expresividad y la formalidad de lenguajes de especificación orientados a modelos, como VDM [7]. Este lenguaje está diseñado para ser utilizado en una amplia variedad de herramientas [4, 5], las cuales soportan diseño por contratos (DBC), chequeo de *assertions* en tiempo de ejecución, comprobación estática, verificación formal a través de las demostraciones de teoremas, entre otras. En las siguientes secciones se introducen conceptos del lenguaje, restricciones y aplicaciones.

2.1. Especificación de la interfaz de comportamiento

El JML describe dos aspectos importantes de un módulo Java. Por un lado su interfaz, la cual consiste en los nombres y la información estática que se encuentran en las declaraciones, y por otro lado describe su comportamiento, que expresa cómo el módulo actúa cuando se ejecuta. Las especificaciones del JML pueden escribirse como expresiones lógicas denominadas *assertions*, las cuales se tratan como comentarios en un compilador de Java, y se escriben entre '/*@ ... @*/' o después de '//@'. Los comentarios de una sola línea '//' son ignorados por Java y por el JML. Estas especificaciones pueden escribirse como *assertions* en la misma clase que se quiera especificar, como en la Figura 2.2, o en ficheros distinto a éste. Para este último caso hay que escribir en dicha clase la cláusula ***refine*** seguido del nombre del fichero de especificación, como se muestra en la Figura 2.1.

```
//@ refine "Persona.jml-refined";
//@ model import org.jmlspecs.models.*;
public class Perona {
   ...
}
```

Figura 2.1: Cláusula Refine

La sintáxis de las declaraciones de Java están permitidas en el JML, como por ejemplo declarar un nombre público, tipo de retorno y parámetros de entrada de un método, entre otras. La información sobre el comportamiento se especifica en una *assertion*. Dichas especificaciones se escriben antes de la cabecera del método o entre la cabecera del método y su cuerpo.

2.1.1. Especificación de métodos

En el JML las especificaciones de los métodos pueden contener pre y post-condiciones. Las especificaciones de las pre-condiciones son evaluadas antes de la ejecución del cuerpo del método, mientras que las especificaciones de las post-condiciones son evaluadas después que finaliza la ejecución del mismo. Además también existen las especificaciones de estado interno llamadas invariantes (a detallarse en la sección 2.2.3), las cuales se verifican durante la ejecución del método.

Las pre y post-condiciones en el JML se escriben utilizando expresiones extendidas de Java. Para definir pre-condiciones, se utiliza la cláusula ***requieres***, en cambio para las post-condiciones, se utiliza la cláusula ***ensures***. En el capítulo anterior se definió el contrato de software del método *SetEdad* de la clase *Persona*, en la Figura 2.2 se especifica dicho contrato utilizando el JML.

```
public class Persona {
   ...

   /*@ requires valor >= 0;            //(1)
   @ ensures this.edad >= 0 &&         //(2)
   @ this.edad == valor;               //(3)
   @*/
   public void setEdad(int valor) {
      ...
   }
}
```

Figura 2.2: Especificación de pre y post condición

En *(1)* se especifica la pre-condición: El parámetro de entrada tiene que ser mayor o igual a 0. En *(2)* se especifica la post-condición: El atributo *edad* es mayor o igual a 0. En *(3)* se especifica la post-condición: El atributo *edad* tiene el mismo valor que el parámetro *valor*.

Una post-condición puede tener un comportamiento normal, como en el ejemplo anterior, o excepcional. Para este último caso, la cláusula *signals* se especifica para que atrape la excepción, por ejemplo si la edad es menor a 0, como se muestra en la Figura 2.3.

```
public class Persona {
   ...

   //@ signals (IllegalArgumentException e) valor < 0;
   public void setEdad(int valor)
                    throws IllegalArgumentException{
      ...
   }
}
```

Figura 2.3: Especificación excepcional

2.1.2. Especificaciones de varios comportamientos

Se puede especificar en una post-condición más de un comportamiento. Generalmente un comportamiento normal se indicará con la cláusula ***normal_behavior***, en caso que solo haya un comportamiento

puede omitirse dicha cláusula como en el ejemplo anterior; esto significa que cuando la pre-condición se cumple, la invocación al método debe retornar normalmente, sin lanzar una excepción. Se pueden combinar varios comportamientos utilizando la cláusula ***also*** para unirlos. Para el ejemplo anterior, *setEdad*, se especifica los siguientes comportamientos normales.

1. Si el parámetro de entrada es mayor o igual a 0, el atributo *edad* es igual al parámetro de entrada.

2. Si el parámetro de entrada es menor a 0, el atributo *edad* es igual a 0.

Cada comportamiento diferente se indica con la cláusula *normal_behavior* unidos por la cláusula *also*. Lo anterior queda ejemplificado en la Figura 2.4.

```
public class Persona {
   ...

   /*@ public normal_behavior
   @ requires valor >= 0;
   @ ensures this.edad == valor;
   @ also
   @ public normal_behavior
   @ requires valor < 0;
   @ ensures this.edad == 0;
   @*/
   public void setEdad(int valor) {
      ...
   }
}
```

Figura 2.4: Especificaciones normales de varios comportamientos

Para especificar un comportamiento excepcional generalmente se utilizará la cláusula ***exceptional_behavior*** y si el método devuelve una excepción, se genera la misma usando ***signals***. A continuación se ejemplifica un comportamiento normal y otro excepcional en lenguaje natural para el método *setEdad*.

1. Si el parámetro de entrada es mayor o igual a 0, el atributo *edad* es igual al parámetro de entrada.

2. Si el parámetro de entrada es menor a 0, se lanza una excepción.

En la Figura 2.5 se muestran las especificaciones en JML.

```
public class Persona {
   ...

   /*@ public normal_behavior
   @ requires valor >= 0;
   @ this.edad == valor;
   @ also
   @ public exceptional_behavior
   @ requires valor < 0;
   @ signals (IllegalArgumentException e) true;
   @*/
   public void setEdad(int valor) {
      ...
   }
}
```

Figura 2.5: Especificaciones normales y excepcionales de varios comportamientos

2.1.3. Especificación de ejemplos

El JML permite incluir ejemplos como parte de la especificación, mediante la cláusula ***for_example***. Los ejemplos en ocasiones brindan una mayor claridad a la especificación. Un ejemplo de su utilización se muestra en la Figura 2.6.

```
public class Persona {
   ...

   /*@ public normal_behavior
   @ requires valor >= 0;
   @ ensures this.edad >= 0 &&
   @ this.edad == valor;
   @ for_example
   @ requires valor == 3;
   @ ensures this.edad >= 0 &&
   @ this.edad == 3;
   @*/
   public void setEdad(int valor) {
      ...
   }
}
```

Figura 2.6: Especificación de ejemplo

2.1.4. Especificaciones de peso ligero y peso pesado

En el JML no se está obligado a especificar todo el comportamiento de un método. En efecto, este lenguaje presenta dos estilos de especificación de métodos, **especificación de peso ligero** y **especificación de peso pesado**. Para el primer estilo, el usuario sólo especifica lo que es de su interés. Por otro lado, en un caso de especificación de peso pesado, el JML espera que el usuario defina

en detalle el comportamiento, pudiendo omitirse solo aquello que se considere irrelevante. Se puede distinguir entre estos dos estilos utilizando diferente sintáxis [6], como por ejemplo en especificaciones ligeras no se utilizan las cláusulas ***normal_behavior*** o ***exceptional_behavior***.

2.1.5. Especificaciones informales

Ya sea por las limitaciones del lenguaje o para simplificar la escritura de contratos del implementador, existen comentarios que permiten descripciones informales en las especificaciones, los cuales se defininen mediante *(* *)*. El JML trata a la descripción informal como una expresión *booleana* que siempre se evalúa en *true*, permitiendo unir especificaciones formales con especificaciones informales a través del operador '&&'. Por ejemplo, la especificación del JML mostrada en la Figura 2.7, describe el comportamiento del método *setEdad* utilizando descripciones formales e informales.

```
public class Persona {
   ...

   /*@ requires valor >= 0;
   @ ensures (* el atributo edad
   @ es asignado *) &&
   @ this.edad == valor;
   @*/
   public void setEdad(int valor) {
      ...
   }
}
```

Figura 2.7: Especificaciones informales

2.2. Formalización del lenguaje

En esta sección se formalizan los conceptos más importantes del JML.

2.2.1. Expresión de especificación

Una expresión de especificación es similar a un predicado de lógica que se evalúa a verdadero o falso. Permite describir condiciones que deben cumplirse en determinados puntos durante la ejecución de un programa. Además se puede especificar el comportamiento de métodos y clases. Estas expresiones se pueden especificar con una condición que utilice:

- Operadores de comparación
- Operadores lógicos
- Cuantificadores
- Operadores adicionales

Operadores de comparación

Las expresiones que se pueden utilizar como predicados en el JML son una extensión de expresiones Java como '==', '*!*=', '<', '<=', etc; las cuales no pueden contener operadores que modifiquen variables como los operadores de asignación '=','+=','-=', y los operadores de incremento y decremento '++','--'.

Operadores lógicos

En el Cuadro 2.1 se muestran algunos operadores lógicos utilizados en las especificaciones.

Sintáxis	Significado
==>	Implicación
<==	Contra implicación
<==>	Equivalencia
<=!=> b	No equivalencia
&& b	Operador AND
|| b	Operador OR
! b	Operador NOT

Cuadro 2.1: Descripción de algunos operadores lógicos

Dado un método en la clase *Persona*, que devuelve verdadero si la persona es mayor de edad y falso en caso contrario, se puede especificar el método utilizando el operador lógico '<==>'. En la Figura 2.8 se muestra la cláusula *result* que almacena el resultado del método, siendo el tipo de dato el mismo que se retorna en el método.

```
public class Persona {
   ...

   //@ ensures \resut <==> this.edad > 18;
   public boolean esMayorDeEdad() {
       ...
   }
}
```

Figura 2.8: Ejemplo de uso de Operadores Lógicos

Cuantificadores

Todos los cuantificadores tienen la misma sintáxis; definición de una variable ligada, rango de la variable y expresión sobre la que se aplica el cuantificador. En el Cuadro 2.2 se muestran algunos cuantificadores utilizados en el JML.

Sintáxis	Significado
\forall	Para todo
\exists	Existe
\sum	Suma
\product	Producto
\num_of	Número de
\max	Máximo
\min	Mínimo

Cuadro 2.2: Descripción de algunos Cuantificadores

Sea la clase *Adulto*, que hereda de la clase *Persona* implementada anteriormente, con atributos *profesion* e *hijos*, las funciones *getters* y los métodos *setters* de dichos atributos. En la Figura 2.9 se especifica el método *setHijos*, para mostrar un ejemplo concreto del cuantificador *forall*.

```
public class Adulto extends Persona {
    private String profesion;
    private Persona[] hijos;
    ...

    /*@ requires lista != null;              //(1)
    @ ensures (\forall int j;                //(2)
                0 <= j && j < lista.length;
    @              lista[j].edad >= 0);
    @*/
    public void setHijos(Persona[] lista) {
       ...
    }
}
```

Figura 2.9: Cuantificadores

En *(1)* se especifica la pre-condición: El parámetro de entrada tiene que ser distinto de *null*. En *(2)* se especifica la post-condición: Para todos los hijos, la edad tiene que ser válida (mayor o igual a 0).

Operadores adicionales

El JML tiene algunos operadores adicionales que facilitan la especificación. Los mismos pueden ser utilizados en condiciones tanto normales como excepcionales. En el Cuadro 2.3 se muestran algunos operadores que pueden ser utilizados en el JML.

Sintáxis	**Significado**
\result	Valor devuelto por el método
\old(E) y \pre(E)	Valor que toma la expresión E al comenzar la ejecución del método. La cláusula *old* se puede utilizar en post-condiciones normales y excepcionales, mientras que *pre* solo puede ser usada en *asserions* que se encuentren dentro del cuerpo del método
\not_assigned(X1,...,Xn)	Especifica que sólo un subconjunto de grupos de datos *(X1,...,Xn)* no deben ser asignados en la ejecución del método.
\not_modified(X1,...,Xn)	Especifica los campos *(X1,...,Xn)* cuyos valores deben permanecer sin modificaciones durante la ejecución del método.
\only_accessed(X1,...,Xn)	Especifica que sólo un subconjunto de grupos de datos *(X1,...,Xn)* deben ser leídos en la ejecución del método.
\only_called(M1,...,Mn)	La ejecución del método sólo se llama desde el subconjunto de métodos *(M1,...,Mn)*.
\only_captured(X1,...,Xn)	La ejecución del método sólo captura referencias a partir de un subconjunto de los grupos de datos *(X1,...,Xn)*

\fresh(X1,...,Xn)	El subconjunto de datos *(X1,...,Xn)* no son nulos y no han sido asignados antes de la invocación al método, sino que seán asignados en él.
\duration	Especifica el tiempo máximo que puede durar la ejecución del método
\space	Describe la cantidad de espacio de almacenamiento dinámico, en bytes, asignado al objeto
\working_space	Describe la cantidad máxima especificada del espacio heap, en bytes, utilizado por la llamada al método
\reach(X)	Permite hacer referencia a un conjunto de objetos accesibles desde algún objeto en particular. Por ejemplo, *\reach(X)* denota el conjunto más pequeño, *JMLObjectSet*, que contiene el objeto denotado por *X*. Si *X* es nula, entonces el conjunto es vacío, de lo contrario, contiene *X*, todos los objetos accesible a través de todos los campos de *X*, todos los objetos accesibles a través de todos los campos de estos objetos y así sucesivamente, de forma recursiva.
\nonnullelements	Especifica que un *array* y sus elementos no son nulos
\typeof(E)	Devuelve el tipo dinámico más específico del valor de la expresión *E*. Si *E* es *null*, entonces el resultado no se especifica, pero si *E* tiene un tipo estático que es un tipo de referencia,entonces *\typeof(E)* significa lo mismo que E.getClass().
\elemtype(T)	Devuelve el tipo estático más específico compartido por todos los elementos de *T*. Por ejemplo, \elemtype(\TYPE(int [])) es \TYPE(int). El argumento de *\elemtype* debe ser una expresión de tipo *TYPE*, que el JML considera ser el mismo que *java.lang.Class*. Si el argumento no es un tipo de matriz, entonces el resultado es nulo.
\type(T)	Este operador puede ser utilizado para introducir literales de tipo *TYPE* en las expresiones. Una expresión de la forma *\type(T)*, donde *T* es un nombre de tipo, tiene el tipo *TYPE*. Dado que en el JML *TYPE* es el mismo que *java.lang.Class*, una expresión de la forma *\type(T)* significa lo mismo que E.class
\lockset(E)	Denota el conjunto de bloqueos mantenidos por el subproceso actual. Es del tipo *JMLObjectSet*.
\is_initialized(T)	Devuelve verdadero sólo cuando su argumento de tipo de referencia, *T*, es una clase que ha terminado su inicialización estática.
\invariant_for(E)	Devuelve verdadero sólo cuando su argumento satisface el invariante de su tipo estático, por ejemplo *\invariant_for*((*MyClass*) o) es verdadera cuando o satisface el invariante de *MyClass*. En secciones posteriores se describirá el concepto de *invariante*

\lblneg y \lblpos	Expresiones que se utilizan para fijar etiquetas a las expresiones, las cuales pueden ser impresas en varios mensajes, por ejemplo, para identificar a una afirmación que ha fallado. Tal expresión tiene una etiqueta y un cuerpo.
\assert(P)	Comprueba que el predicado especificado P es cierto en el punto dado en el programa.
\assume(P)	Asume que el predicado P es verdadero.

Cuadro 2.3: Descripción de algunos operadores del JML

Para mostrar algunos ejemplos de estos operadores consideramos el método *incrementarPeso*, perteneciente a la clase *Persona*, de la Figura 2.10.

```
public class Persona {
   private String nombre;
   private double peso;
   private int edad

   /*@ requires valor >= 0;                          //(1)
   @ ensures (\result == \old(peso) + valor) &&      //(2)
   @ \not_assigned(edad, nombre) &&                  //(3)
   @ \modified(peso) &&                              //(4)
   @ \result == peso;                                //(5)
   @*/
   public int incrementarPeso(int valor) {
       ...
   }
}
```

Figura 2.10: Operadores del JML

En *(1)* se especifica la pre-condición: El parámetro de entrada 'valor' tiene que ser mayor a 0. En *(2)* se especifica la post-condición: El valor devuelto por el método es el valor que tenía el atributo 'peso' antes de invocarlo, sumado el parámetro de entrada 'valor'. En *(3)* se especifica la post-condición: Los atributos 'edad' y 'nombre' no son asignados durante la ejecución del método. En *(4)* se especifica la post-condición: El atributo 'peso' es modificado. En *(5)* se especifica la post-condición: El atributo 'peso' es igual al valor devuelto por la función.

2.2.2. Modificadores

Además de los modificadores de Java, en el JML se pueden utilizar los modificadores propios del lenguaje. A continuación se presentan dichos modificadores.

Modificadores de Privacidad y Visibilidad

Las especificaciones públicas sólo pueden hacer referencia a variables o métodos que sean accesibles por el usuario, como son los atributos y métodos públicos de la clase, y los argumentos del propio método. Además de estas especificaciones también es posible realizar especificaciones protegidas y privadas. Para el primer caso, se puede incluir no sólo los métodos y variables públicos sino también los protegidos. En el caso de las especificaciones privadas se le agrega además los métodos y variables privados, estando

dirigidas al propio programador, con el doble objetivo de completar la especificación y de realizar una comprobación más completa de la implementación.

Las especificaciones públicas se indican en el JML con la cláusula ***public***, las protegidas con la cláusula ***protected*** y las privadas con ***private***. Por defecto, el JML supone que se trata de una especificación pública.

Especificaciones públicas y protegidas

El modificador ***spec_public*** permite declarar una variable o método como una especificación pública.

El modificador ***spec_protected*** permite declarar una variable o método como una especificación protegida. Al igual que el modificador anterior las mismas pueden ser declaradas cuando tienen una visibilidad más restrictiva en Java. En este caso tienen que tener acceso privado o acceso por defecto (visibilidad del paquete).

Por ejemplo para que se pueda acceder desde la clase *Adulto* a atributos de la clase *Persona*, si estos son privados se deben de especificar como públicos o protegidos.

```
public class Persona {
   private /*@ spec_protected @*/ int edad;
}
```

Desde la clase *Adulto*, como hereda de la clase *Persona*, se puede usar en la especificación el atributo *edad* ya que el mismo es protegido.

Assertion no nula

La aserción no nula permite que un campo nunca sea nulo. Se indica empleando la cláusula ***non_null***. Por ejemplo el nombre de una persona no puede ser nulo.

```
public class Persona {
   private /*@ non_null @*/ String nombre;
}
```

Métodos Puros

En una especificación está permitido invocar a métodos siempre que estos no alteren variables no locales del método. El JML llama puros, *pure*, a este tipo de métodos o constructores, y requiere que contengan el modificador ***pure*** en su declaración. Este modificador se debe añadir junto al resto de modificadores de Java, por ejemplo luego del modificador *public*.

```
public /*@ pure @*/ boolean esMayorDeEdad(){...}
```

En la Figura 2.11 se puede observar el método *getHijosMayorDeEdad* perteneciente a la clase *Persona*, que devuelve todos los hijos mayores de edad y utiliza el método puro *esMayorDeEdad* en la especificación. Un método también es puro cuando no es estático y aparece en la especificación de una clase o interfaz pura. Del mismo modo, un constructor también es puro si aparece en la especificación de un clase pura.

```
/*@ requires lista != null;
@   assignable \nothing
@   ensures (\forall int j; 0 <= j && j < \result.length;
@      \result[j].esMayorDeEdad());
@*/
public Persona[] getHijosMayorDeEdad(Persona[] lista) {
 ...
}
```

Figura 2.11: Ejemplo de uso del Modificador *Puro*

Modelo

Se utiliza el modificador modelo, *model*, para especificar variables y métodos en el JML. Los mismos se utilizan en *assertions* y no en código Java. Las variables no pueden ser asignadas, solo cambia su valor cuando cambia su representación. Se especifica utilizando la cláusula ***model***. En la Figura 2.12 se especifica el método *setEdad* de la clase *Persona* utilizando *model*.

```
public class Persona {
  ...
  private int edad;
  ...

  //@ public model boolean edadValida;
  //@ represents edadValida <-- (edad >= 0);

  //@ requires valor >= 0;
  //@ assignable edad;
  //@ ensures edadValida && this.edad == valor;
  public void setEdad(int valor) {
     ...
  }
}
```

Figura 2.12: Modificador *Modelo*

Fantasma

Una variable fantasma, *ghost*, también está presente sólo para los propósitos de especificación y por lo tanto no se puede utilizar fuera de las *assertions*. Sin embargo, a diferencia de una variable modelo, su valor es directamente determinado por su inicialización o por un conjunto de declaraciones. Se puede asignar una variable fantasma utilizando la cláusula ***set***

Aunque las variables *ghost* y *model* sólo se aplican para las especificaciones, el JML utiliza el mismo *namespace* para esos nombres como los utiliza Java. Por lo tanto, no se puede declarar una variable que sea a la vez modelo o fantasma y además una variable Java, en la misma clase. Lo mismo ocurre con el nombre de un método. En la Figura 2.13 se utiliza el ejemplo anterior utilizando una variable fantasma.

```
public class Persona {
  ...
  private int edad;
  ...
  //@ public ghost boolean edadValida = false;

  //@ requires valor >= 0;
  //@ assignable edad;
  //@ ensures this.edad == valor;
  public void setEdad(int valor) {
      if (valor < 0) {
         this.edad = 0;
        //@ set edadValida = false;
      }
      else {
        this.edad = age;
        //@ set edadValida = true;
      }
  }
}
```

Figura 2.13: Modificador *Fantasma*

Instancia

El modificador instancia, *instance*, indica que un campo no es estático. Se especifica utilizando la cláusula ***instance***.

Asistente

Normalmente una invariante (sección 2.2.3) se aplica a todos los métodos de una clase o interfaz, sin embargo quedan exentos aquellos métodos privados o constructores declarados con el modificador asistente, *helper*. Tales métodos auxiliares se pueden utilizar para evitar la duplicación de código.

Seguimiento

El modificador de seguimiento, *monitored*, puede ser utilizado al declarar un campo que no es modelo, para que un hilo deba mantener el bloqueo en el objeto que contiene el campo, antes de leerlo o escribirlo [8].

Sin inicializar

El modificador sin inicializar, *uninitialized*, se aplica a un campo o una declaración de variables que a pesar de estar inicializado, la ubicación tiene que ser considerada sin inicializar. Por lo tanto, el campo o variable debe ser asignado en cada camino antes de ser leído [8].

2.2.3. Tipo de Especificaciones

En esta subsección se describe la forma en que el JML especifica los tipos de datos abstractos (TDAs).

Invariante

Es una propiedad que debe mantener el mismo estado visible para todos los clientes. El JML distingue dos tipos de invariantes: de bucle y de clase.

Invariante de bucle

Predicado cierto durante todas las iteraciones del bucle. Además de la invariante se debería de definir una cota, la cual asegura que el bucle finaliza en algún momento. Invariante del bucle: cláusula ***maintaining***. Cota del bucle: cláusula ***decreasing***. Estas expresiones deben indicarse antes del bucle y son válidas para todos los tipos de bucles de Java, como se muestra en la Figura 2.14.

```
/*@ maintaining 0 <= i && i<= n;
@ decreasing n-i;
@*/
for (i=0; i<n; i++)
{
    //código
}
```

Figura 2.14: Invariante de bucle

Invariante de clase

Predicado que expresa propiedades que son ciertas durante toda la vida de los objetos de una clase. Dichos predicados deben ser verdaderos cuando el objeto es estable. Un objeto es inestable cuando se está ejecutando alguno de sus métodos. Por esta razón la comprobación de la invariante se realiza antes y después de la llamada a un método, y por tanto se comporta como una post-condición común a todos los métodos de la clase. Se especifica mediante la cláusula ***invariant***, puede incluirse en cualquier parte de la clase excepto dentro de un método. En la Figura 2.15 se muestra la declaración de una invariante de clase, en el ejemplo esta propiedad debe de cumplirse antes y después de invocar todos los métodos pertenecientes a la clase *Persona*, como por ejemplo el método *setEdad*, por lo que no es necesario incluirla en la post-condición de cada métodos.

```
public class Persona {
   ...
   private int edad;
   ...

   //@ invariant edadValida <==> (edad >= 0);

   //@ requires valor >= 0;
   //@ assignable edad;
   //@ ensures this.edad == valor;
   public void setEdad(int valor) {
      ...
   }
}
```

Figura 2.15: Invariante de clase

Constraints

History Constraints, conocidos como *Constraints*, se utilizan para especificar la forma en que los objetos pueden cambiar los valores de un estado a otro (es decir, pre-estado a un post-estado) durante la ejecución del programa. En el JML, se utiliza la cláusula *constraint* para especificar dicho comportamiento. Generalmente se escribe usando la cláusula *old* para relacionar el estado anterior con el estado actual, ya que son predicados de dos estados. Por otro lado, *invariantes* (visto previamente) no puede utilizar expresiones con cláusula *old*, porque son predicados de un solo estado. Sea el método *agregarHijo* de la clase *Adulto*, el cual incrementa la variable *cantHijos* en 1. La post-condición indica que la cantidad de hijos luego de invocado el método es igual a la cantidad de hijos antes de la invocación más 1, como se muestra en la Figura 2.16.

```
public class Adulto {
   ...
   private int cantHijos;

   //@ constraint this.cantHijos == \old(this.cantHijos) + 1;

   public void agregarHijo(Persona persona){
      ...
   }
}
```

Figura 2.16: *History Constraints*

Para este caso, si la restricción falla, se muestra un error indicando la violación.

2.2.4. Grupos de datos

El JML permite la creación de grupo de datos, *datagroups*. A continuación se detalla un ejemplo para mostrar la importancia de los mismos. Se quiere agregar la fecha de nacimiento a la clase *Persona*, para ello se agregan los atributos año, mes, día y un método *SetFechaNacimiento* que setea la fecha de nacimiento de la persona, como se muestra en la Figura 2.17.

```
public class Persona {
   ...
   protected /*@ spec_protected @*/ int anio, mes, dia;
   ...

   /*@ requires ...;
   @ assignable anio, mes, dia;
   @ ensures ...;
   @*/
   public SetFechaNacimiento(int anio, int mes, int dia) {
       ...
   }
}
```

Figura 2.17: *Cláusula assignable*

La cláusula ***assignable*** de esta forma presenta dos inconvenientes, el primero es que se expone en detalle la implementación, al mencionar los nombres de los campos protegidos, y la segunda es que la especificación es restrictiva para cualquier subclase futura. *Datagroups* provee una solución para este problema utilizando *JMLDataGroup*. Como se puede observar en la Figura 2.18, la especificación declara público al grupo de datos fecha y privado a los tres campos pertenecen a este grupo de datos. Esto evita tener que exponer los nombres de los campos y las subclases podrán ampliar el grupo de datos con campos adicionales.

```
public class Persona {
   //@ public model JMLDataGroup fechaNac;
   protected int anio; //@ in fechaNac;
   protected int mes; //@ in fechaNac;
   protected int dia; //@ in fechaNac;

   /*@ requires ...;
   @ assignable fechaNac;
   @ ensures ...;
   @*/
   public SetFechaNacimiento(Date fecha) {
       ...
   }
}
```

Figura 2.18: *Grupo de datos*

Los *datagroups* pueden anidarse de la siguiente manera.

```
public model JMLDatagroup fecha; //@ in objectState;
```

ObjectState es el *datagroup* por defecto definido en *Object.java*.

2.2.5. Herencia

Al igual que existe la herencia en Java, también se pueden heredar las especificaciones escritas en el JML. Para ello es necesario informar a la especificación de que existe otra en una superclase o interfaz situada en un nivel superior en la jerarquía de herencia. Esto se logra en el JML con la cláusula ***also*** al comienzo de la especificación.

En la Figura 2.19 se muestra la herencia con el método *agregarHijo*, donde en la clase *Persona* se tiene como *pre-condición* que la persona no sea *null* y como *post-condición* que la edad sea válida. Por otro lado, en la clase *Adulto* se guardan los hijos, por lo que se podría agregar como post-condición que la cantidad de hijos haya incrementado en 1. Si bien esta especificación es muy escueta, sirve para entender el concepto de la herencia.

```
public class Persona {
   ...

   //@ requires persona != null;
   //@ ensures persona.edad > 0;
   public abstract void agregarHijo(Persona persona){
     ...
   }
}

public class Adulto extends Persona{
   ...
 private int cantHijos;

  /*@
    @ also
    @ cantHijos == \old(cantHijos) + 1;
    @*/
    public void agregarHijo(Persona persona){
       ...
    }
}
```

Figura 2.19: Herencia

2.3. Aplicación a problemas de algoritmia

En algoritmia se presentan dos cuestiones fundamentales, resolver algorítmicamente un problema y obtener soluciones lo más eficientes posible. Con el JML, además de poder especificar el comportamiento de un algoritmo, se puede acotar el tiempo de ejecución y el espacio en memoria del programa. Para esto se utilizan las cláusulas ***duration*** y ***working_space*** respectivamente. Ambas funcionalidades aún están en fase de desarrollo.

2.4. Restricciones del lenguaje

Como se mencionó anteriormente, en *assertions* no está permitido utilizar operadores que modifiquen variables, tampoco se puede invocar métodos que alteren variables no locales, métodos con efectos laterales. No se puede especificar de manera formal los parámetros de entrada y salida, por lo tanto, métodos que escriben y leen archivos normalmente utilizan las descripciones informales para describir su comportamiento.

2.5. Estado y planes para el JML

Al estudiar el lenguaje se ha observado que la semántica de algunas características incluidas en este documento, son todavía objeto de investigación. Esto es consecuencia de que el JML está todavía en desarrollo y que puede sufrir variaciones notables. El JML no dispone de ningún entorno amigable en el que se pueda trabajar, resultando complicado localizar un error sintáctico en el lenguaje. Normalmente es utilizado con herramientas como Eclipse, EditPlus, entre otras, las cuales facilitan la tarea de compilación y ejecución. Por esa razón se necesita desarrollar, como aporte al proyecto de Leavens [9], un IDE para el JML. Otra área de trabajo futuro para el JML es la concurrencia. Se ha investigado el uso de atomicidad

para especificar multi-hilo en programas Java; sin embargo, estas ideas no están soportadas en la mayoría de las herramientas del JML y su uso no se ha explorado completamente.

Capítulo 3

Herramientas

Como se estudió en el capítulo anterior, Java Modeling Language (JML) es un lenguaje de especificación formal utilizado por programadores, los cuales requieren de herramientas capaces de compilar y ejecutar las especificaciones realizadas. Además dichas especificaciones pueden no ser sencillas por lo que resulta necesario contar con herramientas que asistan a la hora de especificar programas, de modo que el esfuerzo requerido para dicha tarea sea el menor posible. Existe una gran variedad de herramientas capaces de facilitar la escritura, chequeo, documentación, compilación y prueba de los distintos módulos que se necesita especificar.

En este capítulo se describen las distintas herramientas encontradas distribuidas en categorías. Se estudiaron dos categorizaciones propuestas en otros trabajos[12, 13]. Dado que las categorizaciones anteriores no contemplaban todas las herramientas existentes y algunas categorías necesitaban ser divididas, se propuso una nueva categorización a partir de ellas.

3.1. Categorizaciones existentes de herramientas

Diferentes autores exponen distintas formas de agrupar las herramientas, es el caso de Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens y Erik Poll (Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2)[12] los cuales presentan dos grandes categorías:

1. Chequeo en tiempo de ejecución (RAC).
2. Verificación estática.

Estas categorías son dos formas complementarias de realizar el chequeo de las *assertions*.

(RAC) refiere al chequeo en tiempo de ejecución del programa, donde cualquier violación es reportada. La idea de chequear contratos en tiempo de ejecución fue popularizado por Eiffel.

En la verificación estática se utilizan técnicas lógicas para probar la correctitud, se prueba que no haya violaciones de las especificaciones antes de ser ejecutado el código. El adjetivo estático da énfasis en que la verificación se realiza mediante un análisis estático del código sin necesidad de ejecutar el mismo.

RAC y Verificación Estática tienen fortalezas complementarias. La verificación estática en comparación con RAC a menudo provee una mejor verificación, ya que recorre todos los caminos posibles de ejecución y prueba que las especificaciones se cumplan en ellos. Sin embargo estas ventajas tienen un precio, las herramientas de Verificación Estática generalmente requieren especificaciones más completas, no solo para el modulo a ser chequeado, sino también para los módulos y librerías de los que depende.

Otra categorización encontrada, más extendida en comparación a la anterior, fue presentada por los autores Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T.Leavens, K. Rustan M. Leino, Erik Poll (An overview of JML tools and applications)[13]. Se proponen agrupar las herramientas en cuatro categorías, las cuales incluyen la categorización antes mencionada. A la primer categoría de la categorización anterior se agrega el testeo de las *assertions* mediante la definición de pruebas. Además, se agregan dos nuevas categorías que hacen referencia a la generación

de las especificaciones mediante herramientas y la documentación del código Java anotado con el JML. Las categorías son:

1. Chequeo en tiempo de ejecución de *assertions* y testeo de las mismas mediante pruebas.
2. Chequeo estático y verificación.
3. Generación de especificaciones.
4. Documentación.

Se realizó una búsqueda de las herramientas existentes vinculadas al JML para cada una de las categorías de esta categorización. A continuación se describen estas categorías, mencionando algunas de las herramientas encontradas para cada una de ellas.

La primer categoría hace referencia, como fue mencionado en la categorzación anterior, al chequeo en tiempo de ejecución de código anotado en el JML. Las herramientas principales disponibles en este caso son JMLC, JML4C y OPENJML entre otras, las cuales permiten compilar el código Java para luego en tiempo de ejecución, chequear las *assertions* utilizando por ejemplo la herramienta JMLRAC. Para realizar el testeo de clases Java anotadas con el JML se dispone de las herramientas JMLUNIT y JMLUNITNG, las mismas son una combinación de chequeo en tiempo de ejecución de *assertions* con testeo unitario. JMLUNING genera casos de pruebas de forma automática partiendo de clases anotadas con el JML.

La siguiente categoría consiste, al igual que la segunda categoría de la categorización anterior, en comprobar la correctitud de un programa sin ejecutar el mismo, sino que probar formalmente su correctitud mediante pruebas. Las herramientas de chequeo estático dependen de cuán bien se ha especificado el programa y cabe destacar que el problema de verificar la correctitud de un programa respecto a una especificación no es decidible. Es necesario también que estas herramientas manejen algún tipo de automatización de decisiones o proveer algún tipo de interacción con usuarios. Las herramientas disponibles para esta categoría son: ESC/JAVA, ESC/JAVA2, LOOP, JACK, KRAKATOA y KEY entre otras.

En tercer lugar se encuentra la generación de especificaciones, las herramientas incluidas dentro de esta categoría refieren a herramientas especializadas en ayudar al programador en la escritura de las especificaciones, ahorrando de esta manera el costo de producción de código anotado con el JML. Ejemplos de estas herramientas permiten: generación de esqueletos de especificaciones (JMLSPEC), búsqueda de anotaciones en clases (HOUDINI) y sugerencias de invariantes (DAIKON).

Por último la documentación. Existen herramientas que facilitan al programador la documentación del código anotado, por ejemplo con documentación en HTML. Esto es útil ya que el código va a ser leído por programadores que necesitan una rápida comprensión del mismo. La herramienta encontrada para este propósito fue JML-DOC.

3.2. Nueva Categorización

Luego de estudiar estas dos propuestas de categorización encontramos conveniente crear un nueva categorización, la cual se basa en la segunda categorización mencionada. De la misma mantuvimos las categorías 2, 3 y 4, y la primer categoría fué dividida en dos categorias. Por un lado, el chequeo en tiempo de ejecución de *assertions* y por otro, la generación y ejecución de casos de pruebas que verifiquen el cumplimiento de las mismas. Por último, esta nueva categorización requería de una nueva categoría la cual englobara todas aquellas herramientas utilizadas para la compilación de las clases Java anotadas con el JML. La nueva categorización contiene las siguientes categorías:

1. Compilación de clases Java anotadas con el JML.
2. Chequeo en tiempo de ejecución de *assertions*.
3. Generación y ejecución de casos de pruebas.
4. Chequeo estático y verificación.
5. Generación de especificaciones.
6. Documentación.

En las siguientes subsecciones se detallan las características principales de cada herramienta, agrupadas por las categorías de la nueva propuesta de categorización.

3.2.1. Compilación de clases Java anotadas con el JML.

Luego de realizar la especificación de los módulos utilizando el JML, los mismos deben ser compilados. Dicha compilación puede realizarse utilizando alguna de las herramientas descritas a continuación. El compilador a utilizar debe ser elegido de acuerdo a cada situación.

JMLC (*Runtime assertion checker compiler [Ultima release: Setiembre 2006]*)

Genera *assertions* de las especificaciones escritas en el JML, capaces de ser ejecutadas en tiempo de ejecución. El compilador del JML, llamado JMLC es una extensión del compilador de Java, el mismo compila programas Java anotados con especificaciones escritas en el JML. El código compilado incluye *assertions* chequeadas en tiempo de ejecución, como por ejemplo, chequeo de pre-condiciones, post-condiciones normales, post-condiciones excepcionales e invariantes, entre otros. La ejecución de tales *assertions* es transparente ya que si las mismas no son violadas, y a excepción de la *performance*, el programa no sufre alteración. La transparencia es garantizada además, ya que las *assertions* del JML no pueden tener efectos colaterales, esto quiere decir que la ejecución de las mismas no altera las variables o constantes pertenecientes al programa original. La ejecución de las clases compiladas con el JMLC son tratadas de igual manera que las clases Java, a excepción de que necesita las librerías del JML (JML Runtime Library, jmlruntime.jar) para que puedan ser interpretadas por la máquina virtual de Java[13].

JML4C [Ultima release: Marzo 2010]

JML4C es un nuevo compilador del JML construido sobre la plataforma JDT de código abierto de Eclipse. Algunas de las características de JML4C incluyen (1) mejora en la velocidad de compilación, (2) soporte de Java 5 con características como los genéricos y mejorado de bucles, (3) soporte de clases anidadas, y (4) mejoras en mensajes de errores. Usando JML4C, por lo tanto, se puede verificar clases escritas con el lenguaje Java 5 anotadas con especificaciones escritas en el JML. Este compilador usa una técnica conocida como '*AST meging*' para mejorar la velocidad de compilación. Un resultado experimental demostró que la técnica de la fusión de AST en promedio es alrededor de 1,6 veces más rápido que la estrategia de doble ronda de JMLC, en general, JML4C es tres veces más rápido que JMLC. El nuevo compilador soporta la mayoría de las características de Nivel 0 y Nivel 1 del JML [16].

OpenJML [Ultima release: Mayo 2013]

OpenJML es un conjunto de herramientas capaces de compilar código fuente anotado con el JML(JMLC), es posible ejecutar código compilado con *assertions*(RAC), hacer chequeo estático (ESC) entre otros. Fue creado para correr en un ambiente Java, actualmente la versión 1.7. Este conjunto de herramientas está en continuo proceso de mejora [15].

AJML [Ultima release: Agosto 2011]

AJML es un compilador similar a JMLC, pero a diferencia del mismo, este soporta chequeo en tiempo de ejecución de assertions para Java ME (Java Platform, Micro Edition), esto no es posible con el compilador estándar JMLC. Ademas genera un compilado mas corto, al poseer optimizaciones a nivel del compilador. Con este compilador se aspira a correr programas, compilados con AJML, de forma mas rápida y eficiente[27].

3.2.2. Chequeo en tiempo de ejecución de *assertions*

Esta categoría hace referencia a las herramientas utilizadas para la ejecución de las clases Java compiladas con un compilador del JML utilizado las librerías necesarias, durante su ejecución todas las condiciones que no son satisfechas son informadas al programador.

JMLRAC [Ultima release: Noviembre 2007]

El JML Runtime Assertion Checker (JMLRAC) es una herramienta capaz de verificar las *assertions* generadas con un compilador del JML, el mismo comprueba en tiempo de ejecución que las *assertions*, por las cuales va ejecutando, se cumplan. Esta herramienta es posible encontrarla como una herramienta independiente, o sea que puede ser obtenida sin formar parte de otras herramientas o que dependa de otras para su utilización. Pero por otro lado existe el paquete OpenJML, comentado en la categoría anterior el cual dentro de su *suite* contiene esta herramienta.[13, 15]

3.2.3. Generación y ejecución de casos de pruebas

Mediante las herramientas descritas en esta categoría, es posible generar de forma automática casos de pruebas. Los mismos pueden ser complementados con pruebas propias del programador o modificados para incluir juegos de datos de prueba diferentes a los sugeridos. Luego de producir los casos de pruebas, los mismos pueden ser ejecutados para realizar la verificación.

JMLUNIT - JMLUNITNG [Ultima release: Enero 2012]

La herramienta de testeo JMLUNIT combina el compilador del JML con JUnit. La misma ayuda al programador a escribir código que genere casos de prueba, para verificar la correctitud de las clases Java contra su especificación, y decida si una prueba es exitosa o no, para ello JMLUNIT usa las especificaciones creadas con el JML y las procesa con JMLC para decidir si el código es ejecutado correctamente. JMLUnitNG es una herramienta automatizada de generación de pruebas unitarias de código Java JML-anotado, usando características de java 1.5+, como genéricos, tipos enumerados, mejorado de los bucles, entre otros. Al igual que JMLUnit esta herramienta utiliza *assertions* para realizar las pruebas. Se mejora el JMLUnit original permitiendo una fácil personalización de los datos a utilizar, para cada parámetro de los métodos de las clases bajo prueba, además mediante el uso de reflexión de Java puede generar automáticamente datos de prueba de tipos no primitivos. Las pruebas generadas por JMLUnitNG utilizan el marco de pruebas unitarias TestNG, y se puede ejecutar (suponiendo un CLASSPATH correctamente configurado) por sí mismos o como parte de grandes suites TestNG. Actualmente, JMLUnitNG genera pruebas para JML4 RAC (elaborados por JML4C) y JML2 RAC (elaborados por JMLC). Como sólo JML4 RAC soporta características modernas de Java, las clases bajo prueba no pueden usar esas características al generar pruebas para JML2 RAC[17].

3.2.4. Chequeo estático y verificación

Aquí se hace referencia a la verificación de los módulos sin requerir su ejecución. En este paso es necesario que la especificación realizada en pasos anteriores sea lo mas completa posible.

TACO [Ultima release: Mayo 2010]

TACO es un herramienta de análisis que chequea de forma estática la correctitud de un programa contra su especificación en el JML. TACO traduce código anotado con el JML a JDynAlloy (un lenguaje de representación intermedia) que es luego traducido a DynAlloy. DynAlloy es una extensión al lenguaje de especificación Alloy que describe propiedades dinámicas de los subsistemas usando acciones. Las acciones nos permiten especificar apropiadamente de forma dinámica propiedades, en particular, propiedades respecto a trazas de ejecución. Alloy es un lenguaje que describe estructuras y brinda herramientas para explorarlas [28].

JML

JML es una herramienta utilizada para chequear las especificaciones del JML, la podemos usar como un sustituto más rápido que JMLC en caso de no ser necesario compilar el código. La misma realiza chequeo de tipos y parseo de las clases Java, así como también referencias validas. Esta herramienta es la base para todas las demás herramientas ya que la misma esta incluida en todas ellas.

ESC/JAVA2 [Ultima release: Noviembre 2008]

Esta herramienta, (*Extended Static Checker, escjava2*), puede encontrar posibles errores de implementación muy rápidamente. Es especialmente bueno para encontrar '*null pointer exceptions*', '*array index out of bounds*' y casteos inválidos. Usa las anotaciones del JML para chequear las especificaciones del programa. La verificación realizada por ESC/JAVA2 es modular, el mismo convierte cada método en un conjunto de condiciones para verificar y se basa en la especificación del JML de todos los demás métodos, a los que refiere para crear las condiciones faltantes que necesita para una correcta verificación. Se verifica cada método por separado lo que es menos costoso que verificar la clase por completo, además de ser más factible dada la tecnología actual de mecanismos automáticos de prueba [12, 13, 14].

LOOP [Ultima release: 2001]

La herramienta LOOP traduce la implementación y especificación de clases JAVA y del JML en su semántica de más alto orden lógico. Sirve de *front-end* a un probador de teoremas en donde las propiedades deseadas a verificar entran en juego. La salida de LOOP es una serie de teorías lógicas para los probadores de teoremas PVS e Isabelle. La traducción automática de las clases JAVA tiene ventajas frente a la manual, como por ejemplo, la traducción mediante LOOP se resuelve en pocos segundos, evita la introducción de errores y además con el apoyo de herramientas las teorías generadas pueden ser modificadas para mejorar su eficiencia en las pruebas [13, 19].

JACK [Ultima release: Noviembre 2007]

La herramienta JACK proporciona un entorno para la verificación de programas en Java con anotaciones del JML. Implementa un cálculo de la condición más débil totalmente automatizado que genera obligaciones de prueba (una obligación de prueba es una fórmula matemática a ser probada, para asegurar que un componente es correcto) a partir de fuentes de Java anotadas. Esas obligaciones de prueba pueden ser ejecutadas usando diferentes demostradores de teoremas. Un objetivo importante del diseño de Jack es que es fácil de usar para los desarrolladores de Java normales, que lo utilizan para validar su propio código. Para permitir a los desarrolladores trabajar en un ambiente familiar, Jack se integra como un *plugin* en el IDE de Eclipse. Se oculta la complejidad matemática de los conceptos subyacentes, el mismo ofrece un visor que permite visualizar las obligaciones de prueba conectadas a caminos de ejecución del programa. Por cada obligación de prueba, se resalta el código fuente correspondiente. Por otra parte las hipótesis y tesis se muestran con notación Java - JML [20].

KRAKATOA [Ultima release: Marzo 2013]

Es una herramienta para la certificación de programas JAVA anotados con el JML, que incuye la herramienta WHY (la herramienta WHY toma programas anotados y produce condiciones para su verificación para ser enviadas a demostradores, que son herramientas dedicadas a probar diferentes teoremas) como plataforma para la generación de obligaciones de prueba. KRAKATOA a partir de programas anotados genera condiciones a ser verificadas, estas condiciones pueden ser probadas utilizando herramientas semiautomáticas que utilizan diferentes estrategias de prueba [21].

KEY [Ultima release: Abril 2013]

KEY es una herramienta de desarrollo de software formal que tiene como objetivo integrar el diseño, la implementación, la especificación formal, y la verificación formal de software orientado a objetos con la mayor fluidez posible. El núcleo del sistema es un nuevo demostrador de teoremas con una lógica dinámica de primer orden para Java con una interfaz gráfica fácil de usar. Acepta tanto especificaciones escritas en el JML como en OCL (*Object Constraint Language*) en los archivos fuente de Java. Estos se transforman en teoremas de lógica dinámica y luego se comparan contra la semántica del programa que están definidos en términos de la lógica dinámica. KEY soporta pruebas de corrección interactivos y totalmente automatizado. Intentos de prueba fallidos pueden ser utilizados para una depuración más eficaz o para pruebas de verificación. Puede ser integrado en las herramientas CASE (*Computer Aided Software Engineering*) para extraer especificaciones [22, 23].

JMLForge [Ultima release: Setiembre 2008]

JMLForge, es una herramienta que chequea de forma estática clases Java contra su especificación escrita en el JML. JMLForge es capaz de parsear el mismo conjunto de instrucciones que las herramientas comunes del JML, sin embargo solo interpreta un subconjunto pequeño de estas e ignora algunas características. Actualmente no soporta completamente *generics* de Java 5. Este proyecto no esta siendo mantenido [29].

3.2.5. Generación de especificaciones

Durante la etapa de implementación los programadores pueden hacer uso de las herramientas incluidas en esta categoría, los programadores podrán apoyarse en ellas para generar las especificaciones necesarias para que las mismas sean lo mas completas posibles, siendo esto fundamental para realizar una buena verificación.

JMLSPEC [Ultima release: 2003]

JMLSPEC tiene dos modos de operación, por un lado produce un esqueleto de un archivo de especificación y por otro compara dos archivos de especificación diferente para buscar inconsistencias. Esta herramienta es capaz de generar, a partir de un archivo anotado con el JML, un archivo que refina (extienda las especificaciones) el archivo recibido. Por defecto el nombre del archivo generado es igual al de entrada pero con extensión '*.refines-spec*'. La idea es que el usuario complete este nuevo archivo con especificaciones que no se encuentran en el archivo de entrada. En el modo de comparación JMLSPEC compara los archivos indicados con el archivo Java de mismo nombre o con el archivo '.class', en caso de no encontrar el archivo '.java'[13].

HOUDINI [Ultima release: Noviembre 2001]

HOUDINI sugiere posibles anotaciones dado un archivo con código Java. Se postulan candidatos a ser anotaciones, los cuales son producto de la comparación de campos y *arrays* con, nulos, valores positivos, negativos y cero. HOUDINI elimina los candidatos que no son correctos, llamando repetidas veces a ESC/JAVA para ver cuáles de los candidatos son inconsistentes con el código. Para finalizar se llama por última vez a ESC/JAVA para proporcionar al usuario una lista de *warnings*. HOUDINI usa todas las anotaciones proporcionadas por el usuario ya que forman parte de la entrada al llamar a ESC/JAVA. HOUDINI puede ser visto no solo como un asistente para encontrar y sugerir posibles anotaciones sino también como una herramienta de chequeo estático [13].

DAIKON [Ultima release: Junio 2010]

Esta herramienta infiere de forma dinámica invariantes probables analizando el comportamiento del código en tiempo de ejecución. La detección dinámica corre un programa, observa el valor que el programa computa y luego reporta propiedades que fueron verdaderas durante la ejecución. Al igual que cualquier análisis dinámico, la exactitud para inferir invariantes depende en parte de la calidad e integridad de los casos de prueba, y además otras ejecuciones pueden falsificar algunas de las propiedades indicadas. Sin embargo, DAIKON utiliza el análisis estático, pruebas estadísticas, y otros mecanismos para reducir el número de falsos positivos. Para aplicar DAIKON a un programa, un usuario ejecuta una versión instrumentada del programa para crear un archivo con la traza de los datos del mismo, a continuación, ejecuta DAIKON sobre el archivo de traza de datos para producir probables invariantes [24, 25].

AutoJML [Ultima release: Noviembre 2006]

AutoJML es un generador de especificaciones del JML. Genera especificaciones a partir de otras especificaciones de mas alto nivel, como diagramas de estado en UML o especificaciones de protocolos de seguridad. La salida de este generador es una combinación de un esqueleto de código Java y especificaciones de métodos y clases en el JML. AutoJML toma como entrada especificaciones escritas en formato

XML, de maquinas de estados finitas, sin embargo utilizando XSL como *front-end*. Puede manejar algunos formalismos como especificaciones hechas en Uppaal y diagramas de estados de UML escritas en formato OMG XMI. Se agrego también la posibilidad de aceptar como entrada especificaciones hechas en formato CAPSL. En el año 2006 se agrego un generador de especificaciones para el protocolo SSH [26].

3.2.6. Documentación

Mientras se definen las especificaciones puede ser deseable que el código sea documentado para proveer una descripción adecuada de los módulos, en dicho caso se podría utilizar la herramienta descrita en esta subsección.

JML-DOC [Ultima release: 2002]

Esta herramienta produce páginas HTML que pueden ser navegadas, las mismas contienen la API y las especificaciones para el código Java, manteniendo un estilo similar al obtenido con JAVADOC [13].

Capítulo 4

Eiffel

En este capítulo se presentan las características principales del lenguaje Eiffel. En el mismo se detallan propiedades de su sintáxis y su semántica utilizando ejemplos prácticos para una mejor comprensión de los mismos.

4.1. El lenguaje

Eiffel es un lenguaje de programación orientado a objetos, diseñado por Bertrand Meyer, que propone un método para el desarrollo de sistemas de calidad. Este método, no solo cubre la etapa de implementación de software, sino también otras etapas del desarrollo de software como por ejemplo: análisis, modelado y especificación; diseño y arquitectura; mantenimiento; documentación. Por otro lado, Eiffel puede ser visto como un método que guía a los analistas y desarrolladores durante el proceso de diseño e implementación de software, haciendo foco en la productividad y en la calidad. Los principales factores de calidad considerados por Eiffel son: confiabilidad, reusabilidad, extensibilidad, portabilidad y mantenibilidad[32, 35].

Este capítulo se centra en mostrar las características principales del lenguaje Eiffel y como aporta en la mejora del factor de calidad Confiabilidad. Un producto de software es confiable cuando está libre de errores y ejecuta de manera esperada. La confiabilidad está compuesta por dos elementos: la correctitud, que indica que un programa es correcto si hace lo que se supone que haga y solo eso, y la Robustez, que hace referencia a que un software es robusto cuando actúa de forma aceptable cuando no puede hacer lo que supone que haga. Para abordar este factor Eiffel introduce la noción de diseño por contratos[33].

4.2. Formalización del Lenguaje

En esta sección se presentan las principales características del lenguaje, detallando las diferentes cláusulas disponibles para la implementación de programas y su especificación utilizando un enfoque de diseño por contratos. Se detallan características de su semántica y sintáxis citando ejemplos para una mejor comprensión de las mismas.

4.2.1. Implementación de Clases

Las clases en Eiffel son definidas utilizando la cláusula ***class***, ellas obedecen la forma presentada a continuación en la que se detallarán solo los elementos relevantes, ([...] denota la opcionalidad del componente).

```
[Notas]
Cabezal_Clase
[Genericos Formales]
```

```
[Obsoleto]
[Herencia]
[Creadores]
[Convertidores]
[Caracteristicas]
[Invariantes]
[Notas]
end
```

Cabezal_Clase es el cabezal de la clase el cual es obligatorio y de la forma [Marcador] ***class*** *Nombre*. El marcador es opcional y es utilizado delante de la cláusula *class*, pudiendo tomar los siguientes valores.

1. deferred
2. frozen
3. expanded

A continuación se describen cada uno de ellos:

Clases derivadas

Un método definido como derivado es un método que en su cuerpo contiene la cláusula ***deferred***, este método no es implementado sino que solamente se define su firma y especificación. Cuando una clase hereda (incorpora las características que la clase padre brinda) de una clase derivada e implementa el método derivado, el método se transforma en un método efectivo. Un método efectivo es un método que ya está implementado para su invocación[34]. Una clase que tiene uno o mas métodos derivados es una clase derivada y se define utilizando la cláusula ***deferred***. En la figura 4.1 se muestra un ejemplo de una clase derivada.

```
deferred class PERSONA
 feature
   edad:INTEGER;
   nombre:STRING;
   peso:DECIMAL;
   getEdad (): INTEGER
     do ... end
   getName (): STRING
     do... end
   getPeso():DECIMAL
   do .. end
   setPeso(p:DECIMAL)
   do .. end
   setEdad(e:INTEGER)
   do .. end
   setNombre(n:STRING)
   do ... end
   incremetarEdad()
     do ... end
   agregarHijo(hijo: PERSONA)
     deferred
   end
   ... Otros elementos derivados o efectivos...
end
```

Figura 4.1: Clase Derivada

En este ejemplo la clase *Persona* es derivada y contiene un método derivado llamado *agregarHijo*, dicho método será implementado en alguna clase que herede de esta clase.

Clases congeladas

Un clase congelada, ***frozen***, es un clase que no permite ser padre de ninguna clase, evitando de este modo ser utilizada en alguna herencia. La cláusula utilizada para este tipo de clase es ***frozen***. Además de clases congeladas es posible definir métodos congelados, ellos no pueden ser redefinidos en clases que extiendan de la clase donde fueron definidos. Si se desea que el método *incrementarEdad* de la clase *Persona* sea congelado agregamos la cláusula ***frozen*** delante de la función, como se muestra en la figura 4.2.

```
deferred class PERSONA
 feature
   edad:INTEGER;
   nombre:STRING;
   peso:DECIMAL;

   frozen incremetarEdad()
     do ... end

   ... Otros métodos ...
end
```

Figura 4.2: Clase Congelada

Para definir que una clase llamada *Adulto* no pueda ser padre de ninguna clase, antes de la cláusula ***class*** se agrega la cláusula ***frozen***. A continuación se muestra un ejemplo:

```
frozen class ADULTO
 feature
   ... Atributos ...

   ... Otros métodos ...
end
```

Clases expandida

Una clase expandida, ***expanded***, corresponde a una clase que debe ser utilizada como una entidad y no como una referencia a la clase. En la figura 4.3 se muestra una clase, llamada *Expandida*, ejemplo de este tipo de clase.

```
expanded class EXPANDIDA
  feature
    valor:INTEGER;

  setvalor(i:INTEGER)
    do
      valor := i;
    end

  getvalor():INTEGER
    do
      Result:=valor;
    end
end
```

Figura 4.3: Clase Expadida

Esta clase puede ser utilizada como atributo de otra clase, y al definir un atributo de tipo *Expandida*, se esta indicando que no se guarda una referencia a dicha clase, sino que se almacena como una entidad dentro de la misma.

4.2.2. Características

Una clase se puede representar mediante sus características, ***features***, permitiendo acceder o modificar de ser necesario las mismas. Una característica puede ser un método o un atributo nombrados debidamente con un identificador. Las características se definen en Eiffel mediante la cláusula ***feature***[34]. Una rutina puede ser una función o un procedimiento, una función sin parámetros puede ser considerada como un atributo ya que, al ser invocada, no es necesario agregar paréntesis. En la figura 4.4 se muestra un ejemplo utilizando la clase *Persona*.

```
deferred class PERSONA
 feature
   edad:INTEGER;
 ...
   getEdad (): INTEGER
     do ... end
   ...
end
```

Figura 4.4: Características

Sea una instancia a de la clase Persona, para obtener el atributo *edad* es posible hacerlo de las siguientes maneras:

```
a.getedad
a.edad
```

De este modo se puede observar que la invocación de la función *getEdad* tiene la misma sintáxis, que el acceso a un atributo con dicho nombre.

4.2.3. Herencia

En Eiffel es posible definir una nueva clase como combinación y especialización de clases existentes mediante la clausula ***inherit***. Dicha cláusula permite listar un conjunto de clases de las cuales se heredan

sus características heredables, permitiendo así la herencia múltiple[34]. En la figura 4.5 se muestra como una clase A hereda dos clases B y C.

```
class A
   inherit
     B
   inherit {NONE}
     C
     export {NONE} all end

   feature
     ... Elementos especificos de la clase A ...
   end
```

Figura 4.5: Herencia Multiple

La clase *A* hereda todas las características exportadas por la clase *B* y tiene permitido modificar los valores de los elementos heredados (herencia ajustable). Mediante la cláusula ***inherit* {NONE}**, se define que la clase *A*, hereda todas las características exportadas por la clase *C*. Además, esta cláusula establece que la clase *A* no tiene permitido modificar los valores de los elementos heredados (herencia no ajustable). La cláusula ***export* {NONE} *all end***, define que si una clase arbitraria *D* llegara a heredar de la clase *A*, los elementos heredados de la clase *C* no serán exportados, por lo tanto no serán heredados por *D*[34].

Como la herencia múltiple es permitida, es posible que surjan conflictos de nombres de elementos heredados, para resolver dicho conflicto se puede renombrar los elementos heredados, el siguiente ejemplo muestra dicha funcionalidad.

```
class A inherit
B rename x as x1, y as y1 end
C rename x as x2, y as y2 end
feature...
```

Para comprender mejor la herencia se muestra un ejemplo, en la figura 4.6, utilizando la clase *Adulto* que hereda de la clase *Persona*. De esa manera la clase *Adulto* hereda todos los atributos y rutinas de la clase *Persona*, pero además los complementa agregando nuevos. Se observa como la clase *Adulto* implementa el método ***deferred*** expuesto por la clase *Persona* transformándolo en un método efectivo.

```
class ADULTO inherit PERSONA
 redefine agregarHijo end
 feature
   profesion:STRING;
   hijos:LINKED_LIST[PERSONA];
   getProfesion (): INTEGER
     do ... end
   setProfesion(p:STRING)
   do .. end
   agregarHijo(hijo: PERSONA)
     require
       hijoValido: hijo /= Void
     do .. end
    ensure
       hijoAgregado: validarHijo()
    end
agregarHijos(hijos:LINKED_LIST)
  do ... end
...
end
```

Figura 4.6: Herencia

Además de la herencia, Eiffel permite dar una mayor flexibilidad en las implementaciones al definir genericidad, que consiste en definir tipos genéricos que representan una clase implementada. Dicha flexibilidad permite definir elementos como *LIST[G]* donde G representa cualquier clase implementada. De forma similar se puede indicar un conjunto de clases permitidas de la forma [A,B,C] donde A, B, C son clases[34].

4.2.4. Entidades especiales de Eiffel

Eiffel posee dos entidades especiales, ellas son ***Result*** y ***Current***, la primera hace referencia a la variable que será retornada en todas aquellas funciones en las cuales se retorne un valor, el valor que se asigne a esta variable dentro del cuerpo de la función, será el retornado por ella. La segunda es una referencia a la instancia actual de la clase que se esta ejecutando, por ejemplo si existe una instancia de una clase A que contiene un método m, dentro del método es posible hacer referencia al estado actual de la clase (atributos y rutinas) utilizando la entidad ***Current***[34].

4.2.5. Conformidad

Conformidad es un aspecto importante en Eiffel, el cual determina cuando un tipo de datos puede ser usado en lugar de otro. Este concepto permite asignar una variable y de tipo Y a una variable x cuyo tipo de datos es X. Además, en funciones que reciben por parámetro un tipo de datos X, es posible invocarlas con un parámetro de tipo Y[34]. Este ejemplo muestra la aplicación de esta característica de Eiffel.

Dada la clase llamada *Adulto_Mayor* que hereda de la clase *Adulto* es posible definir lo siguiente:

```
adulto:ADULTO;
adultoMayor:ADULTO_MAYOR;
adultoMayor := adulto;
```

sea una función *funcionX* tal que:
funcionX(adulto:ADULTO);
podemos invocar dicha función de esta manera:

```
a.funcionX(adultoMayor);
```

La conformidad de un tipo con otro se basa en la herencia, donde la condición para que un tipo X conforme a un tipo Y es:

1. La clase base de X debe de ser un descendiente de Y.
2. T:B[Y] conforma A[X] si y solo si B conforma a A e Y a X.
3. Si Y es expandido, en este caso no hay herencia de por medio por lo que X e Y coinciden.

4.2.6. Convertibilidad

Convertibilidad es un mecanismo que permite asignación y pasaje de parámetros en casos en los que la conformidad de tipos no es posible, pero de todas maneras se desea realizar la conversión. Un tipo de datos no puede convertirse en otro tipo y conformarlo, al mismo tiempo. Si deseamos que un tipo X se pueda convertir en un tipo Y, alcanza con definir un método constructor ***make_from_Y*** en la clase X con la cláusula ***convert*** al comienzo de la clase[34].

4.2.7. Herencia Repetida

Dado que en Eiffel es posible heredar de mas de una clase, puede ser posible que dos ancestros de una clase A tengan un padre en común B. Esta forma de herencia se llama herencia repetida y ella permite que una clase herede de una misma clase de diferentes formas. Si dada una clase A se tienen N padres, los cuales tienen un padre en comun B y sean *f1*,...,*fn* características de esos N padres, las cuales tienen la misma característica f como función heredada de partida, entonces se cumplen dos opciones:

1. Cualquier subconjunto de estas características heredadas por A bajo el mismo nombre final en A tiene como resultado una única característica en A.
2. Sean dos características heredadas bajo un nombre diferente, tiene como resultado dos características de A.

4.2.8. Estructuras de Control

Eiffel posee 5 estructuras de control, ellas son: secuencial, condicional, opción de salto múltiple, bucle y una cláusula especial llamada debug. A continuación se comentará cada una de ellas.

Secuencial

Cuando se habla de secuencial se hace referencia, si no hay excepciones durante la ejecución, a la ejecución de cada instrucción de forma ordenada en que fueron definidas, o sea, se ejecuta una instrucción y al finalizar se continúa con la siguiente ejecución[34].

Condicional

El control condicional tiene la siguiente estructura.

```
Conditional => if SECCION_LISTA_THEN [SECCION_ELSE] end
SECCION_LISTA_THEN => {SECCION_THEN elseif ...}+
SECCION_THEN => EXPRESION_BOOLEANA then INSTRUCCIONES
SECCION_ELSE => else instrucciones
```

Si la expresión booleana es verdadera entonces la acción es ejecutar la sección INSTRUCCIONES, de lo contrario si contienen mas de un *elseif* se evalúan los mismos. En caso de contener una sección SECCION_ELSE la acción es ejecutar la sección INSTRUCCIONES, de lo contrario no tiene efecto.

Opción de salto múltiple

La estructura de control opción de salto multiple consiste en listar un conjunto de condiciones a evaluar y para cada una de ellas una acción a tomar, las condiciones son evaluadas una a una en el mismo orden en que fueron definidas hasta que una sea verdadera en cuyo caso se ejecutarán las instrucciones asociadas a esa condición[34].

Bucle

La estructura de control para los bucles es la siguiente:

```
Loop => INICIALIZACION
[INVARIANTE]
condición_SALIDA
CUERPO_LOOP
[VARIANTE]
end
INICIALIZACION => from INSTRUCCIONES
CONDICION_SALIDA => until EXPRESION_BOOLEANA
CUERPO_LOOP => loop INSTRUCCIONES
```

Su función es ejecutar las instrucciones englobadas en la expresión INSTRUCCIONES hasta que la condición de salida expresada con la expresión CONDICION_SALIDA sea verdadera.

Debug

La cláusula ***debug*** permite englobar un conjunto de instrucciones, identificadas con claves, las cuales pueden ejecutarse o no, dependiendo de la habilitación de instrucciones en ***debug*** proporcionada por la herramienta de desarrollo de Eiffel. Esto permite facilmente habilitar o deshabilitar la ejecución de un conjunto de instrucciones mediante dicha habilitación global o discriminada por claves de cada conjunto de instrucciones[34].

```
debug [ "("LISTA_CLAVES ")" ] INSTRUCCIONES end
```

4.2.9. Creación de Objetos

Eiffel posee la cláusula ***create***, definida al comienzo de la clase permite listar métodos que pueden ser invocados para la creación de la clase. Estos métodos se utilizan para realizar inicializaciones necesarias para crear la clase en un estado deseado. Por ejemplo, es común utilizar un método llamado ***make*** con este fin. Se debe tener presente que al momento de la creación de la clase, todos sus invariantes deben cumplirse. Un ejemplo de esta característica se muestra en la figura 4.7.

```
class ADULTO inherit PERSONA create
    make
 redefine agregarHijo end
 feature
   profesion:STRING;
   hijos:LINKED_LIST[PERSONA];

 make(b:INTEGER)
      do
         edad := b;
      end
   ...
   inicializarEdades()
 require
      hijos /= Void;
   do
      hijos.do_all(agent {PERSONA}.setEdad(0));
ensure
         --Chequea que luego de realizar dicha funcion las edades de
         --los hijos sea 0 para cada uno de ellos.
        hijos.for_all
        (agent (x: PERSONA):BOOLEAN do Result := x.getEdad = 0 end)
 end
...
invariant
    hijos.for_all
        (agent (x: PERSONA):BOOLEAN do Result := x.getEdad > 0 end)
end
```

Figura 4.7: Creación de Objetos

En este ejemplo se muestra como la rutina ***make*** es declarada en la cláusula ***create*** y definida en la cláusula ***feature*** de la clase *Adulto*. Al realizar la siguiente instrucción estamos creando una instancia de la clase *Adulto* llamando a la rutina ***make*** definida.

```
adulto:ADULTO;
create adulto.make(30);
```

Si se quiere crear una clase utilizando la rutina de creación por defecto solo es necesario utilizar la cláusula ***create***, como se muestra a continuación:

```
adulto:ADULTO;
create adulto;
```

4.2.10. Comparación y copia de tipos

En Eiffel existen diferentes operadores de comparación, por ejemplo el operador $=$ que compara si dos referencias a objetos son iguales. Si la comparación no involucra referencias el mismo denota la igualdad de objetos. El operador $\tilde{}$ denota la igualdad de objetos. Por otro lado se encuentran las negaciones de estos operadores, ellos son $/=$ y $/\tilde{}$ para $=$ y $\tilde{}$ respectivamente.

Eiffel posee dos funciones que toda clase hereda, las mismas son ***default_is_equal*** y ***is_equals***, la diferencia entre ellos es que la primer función es congelada y no es posible redefinirla. La función ***default_is_equal*** realiza la comparación campo a campo. Por otro lado la versión original de la función ***is_equals*** tiene el mismo comportamiento que la función ***default_is_equal***, se aconseja redefinirla para

poder implementar una condición de igualdad dependiendo de las circunstancias. La comparación en profundidad, o sea, recursivamente por cada objeto alcanzable, es realizada con la función ***is_deep_equal***[34].

Con respecto a la copia de tipos, Eiffel dispone de las funciones ***copy*** y ***twin***. La función ***copy*** es de la forma *x.copy(y)* y el comportamiento de la misma es sobreescribir el objeto *y* por el objeto *x*, o sea que al finalizar su ejecución *y* será una copia de *x*. Por otro lado la función ***twin*** es de la forma *x := y.twin*, el resultado de ejecutar dicha función es la creación de una nueva instancia con el mismo contenido del objeto de tipo *y*, asignada a la variable *x*. Al igual que las funciones de comparacion, Eiffel posee dos funciones de copia en profundidad, las mismas son ***deep_copy*** y ***deep_twin***. Corresponden a la copia en profundidad correspondiente a las funciones ***copy*** y ***twin*** respectivamente[34].

4.2.11. Llamada a características

Eiffel permite llamar a caracteristicas de una clase utilizando el comando ***call***, el cual permite, dependiendo de su parámetro, obtener una expresión o una instrucción. Si el parámetro es un atributo o una función el resultado será una expresión, en caso de ser un procedimiento el resultado será una instrucción[34].

4.2.12. Agentes

Un agente es una forma de definir un objeto que representa un rutina que se encuentra lista para ser llamada. Un agente puede contener argumentos que son seteados en el momento en que el agente es definido o pueden ser provistos al momento de cada llamada al mismo, los primeros son llamados *operandos cerrados* y los segundos *operandos abiertos*. Por ejemplo consideremos una lista llamada *nuestra_lista*, que se desea verificar si sus elementos cumplen una cierta condición definida por una función. Una forma de lograr eso es definiendo un agente que será quién realice dicha operación, el mismo debe recibir por parámetro un elemento de la lista, como se muestra a continuación:

```
nuestra_lista.for_all ( agent (x: INTEGER): BOOLEAN do Result := (x > 0) end )
```

4.2.13. Manejo de excepciones

Durante la ejecución del código es posible que ocurran eventos anormales que pueden ser motivo de errores de *hardware*, errores en operaciones aritméticas o código no escrito de forma adecuada, todos ellos pueden producir una salida anormal. En dichos casos puede ser necesario manejar los errores para su posible tratamiento, para ello Eiffel posee cláusulas que permiten capturar la excepción y ejecutar alguna acción pertinente. Las cláusulas que brinda Eiffel son ***rescue*** y ***retry***, la primera es utilizada dentro de una función que se desea controlar o manejar las posibles excepciones que ella pueda generar. Dentro de esta cláusula se programan las acciones que se desean ejecutar cuando una excepción ocurre. La clausula ***retry*** tiene sentido solo si existe una cláusula ***rescue***. Esta cláusula se ubica al final del bloque de instrucciones definida como ***rescue***, su función es la de volver a ejecutar la rutina que la contiene[34].

4.2.14. Pre-condiciones y Post-condiciones

Eiffel posee expresiones como ***require*** y ***ensure*** que permiten especificar condiciones que deben de cumplirse al comienzo de la ejecución de una rutina y al finalizar la misma respectivamente. Para explicar el comportamiento se utilizará un ejemplo sencillo que consiste en implementar y especificar la clase *PERSONA* utilizando Eiffel como lenguaje de programación y especificación. En lenguaje natural las pre y post condiciones de la clase *PERSONA* son:

1. Pre-condición: La función *setPeso* debe recibir un entero mayor o igual a 0.
2. Post-condición: Al finalizar la ejecución de la función *setPeso*, el valor del atributo *peso* debe ser igual al valor recibido por parámetro.
3. Pre-condición: La función *agregarHijo* debe recibir por parámetro una referencia a una instancia de tipo *Persona* distinta de vacio.

4. Post-condición: Al finalizar la ejecución de la función *agregarHijo*, se debe de comprobar la validez del hijo agregado invocando la función *validarHijo*, que suponemos implementada.

Para especificar el punto número uno se utiliza la siguiente expresión:

```
require
    p >= 0
```

Y para especificar el segundo punto se agrega la siguiente post-condición:

```
ensure
    Current.peso = p;
```

La especificación del punto 3 y 4 se realiza mediante las siguientes expresiones respectivamente:

```
require
      hijoNoVacio: hijo /= Void

ensure
      hijoValido: validarHijo()
```

La clase correspondiente a la solución de dicho ejemplo se muestra en la figura 4.8.

En este ejemplo el método *setPeso* de la clase *Persona* contiene como parte de su implementación la especificación de un contrato. Dicho contrato está compuesto por las obligaciones o pre-condiciones especificadas con la expresión ***require*** y los beneficios o post-condiciones mediante la expresión ***ensure***. Mediante la expresión ***require*** se deja especificado que al momento de comenzar la ejecución de este método el valor del parámetro p de dicho método debe ser mayor o igual a cero, en caso de no ser así pueden suceder dos cosas, la ejecución del método es interrumpida o la misma continua, pero al finalizar el método no es posible garantizar las post-condiciones. Con la expresión ***ensure*** se especifica que, al culminar la ejecucion, el valor del atributo *peso* será igual al valor recibido por parámetro.

Mediante este sencillo ejemplo es posible especificar el comportamiento esperado de un método y de este modo garantizar su correcto funcionamiento bajo ciertas condiciones impuestas en las pre-condiciones.

```
deferred class PERSONA
 feature
   edad:INTEGER;
 nombre:STRING;
 peso:DECIMAL;
   getEdad (): INTEGER
     do ... end
 setEdad(e:INTEGER)
   do .. end
   getNombre (): STRING
     do... end
 setNombre(n:STRING)
   do ... end
 getPeso():DECIMAL
   do .. end
 setPeso(p:DECIMAL)
   require
      p > 0
   do
    peso := p;
ensure
           Current.peso = p;
 end

 incremetarEdad()
      do ... end

   agregarHijo(hijo: PERSONA)
     require
       hijoNoVacio: hijo /= Void
     deferred
    ensure
       hijoValido: validarHijo()
    end
   ... Otros elementos derivados o efectivos...
end
```

Figura 4.8: Pre-condiciones y Post-condiciones

Para lograr especificaciones en las cuales es necesario iterar colecciones, Eiffel permite realizar dichas iteraciones utilizando funciones que recorran la colección chequeando que ciertas condiciones se cumplan para sus elementos, o ejecutando métodos para cada elemento de ella. Un ejemplo de estas funciones es la función ***for_all***, esta realiza una iteración de la colección chequeando que cada elemento de ella cumpla una condición. Dicha condición es chequeada por una función llamada agente, que se pasa por parámetro a la función ***for_all***. La función agente recibe por parámetro un elemento de la colección y produce un booleano que será producto de evaluar el elemento de la colección en dicha función. Por otro lado Eiffel posee otra función de iteración llamada ***do_all***, que es similar a la función ***for_all*** a excepción de que el agente no produce una salida, sino que solo ejecuta una rutina para cada elemento de la colección. En la figura 4.9 se muestra un ejemplo, utilizando la clase *LINKED_LIST* que implementa el TAD Lista encadenada, la misma es una colección ya que hereda de la clase *COLLECTION*. Se utilizarán la funciones ***do_all*** y ***for_all*** con propósitos diferentes.

```
class ADULTO inherit PERSONA
 redefine agregarHijo end
 feature
   profesion:STRING;
   hijos:LINKED_LIST[PERSONA];
   ...
   inicializarEdades()
 require
      hijos /= Void;
   do
      hijos.do_all(agent {PERSONA}.setEdad(0));
ensure
         --Chequea que luego de realizar dicha funcion las edades de
         --los hijos sea 0 para cada uno de ellos.
        registros.for_all
        (agent (x: PERSONA):BOOLEAN do Result := x.getEdad = 0 end)
 end

end
```

Figura 4.9: Pre-condiciones y Post-condiciones con iteración

Como se puede apreciar, la clase *Adulto* posee un atributo llamado hijos cuyo tipo es ***LINKED_LIST***. Este tipo es un tipo propio de Eiffel que implementa una lista encadenda de elementos, esta clase posee funciones de búsqueda, inserción, eliminación e iteración entre otros. Para este ejemplo el tipo ***LINKED_LIST*** es utilizado para almacenar instancias de la clase *Persona*. La clase *Adulto* posee un método llamado *inicializarEdades* que asigna el valor 0 al atributo *edad* de cada elemento de la lista. Este método realiza una iteración de todos los elementos de la lista utilizando la función ***do_all*** antes descrita, y para cada elemento llama a la función *setEdad* de la clase *Prsona*. Como pre-condición, la función *inicializarEdades*, tiene la condición de que el atributo *hijos* no sea vacio. Para especificar la post-condición utiliza la función ***for_all*** para comprobar que cada elemento de la lista, luego de la ejecución del método, tiene asignado el valor 0 en su atributo edad.

Se puede complementar la especificación de una clase definiendo invariantes, que son condiciones que deben cumplirse a lo largo del ciclo de vida de la clase, las mismas se definen mediante la cláusula ***invariant***. En la figura 4.10 se muestra un ejemplo de su uso utilizando la clase *Adulto* mostrado anteriormente.

```
class ADULTO inherit PERSONA
 redefine agregarHijo end
 feature
   profesion:STRING;
   hijos:LINKED_LIST[PERSONA];
   ...
   inicializarEdades()
 require
      hijos /= Void;
   do
      hijos.do_all(agent {PERSONA}.setEdad(0));
ensure
         --Chequea que luego de realizar dicha funcion las edades de
         --los hijos sea 0 para cada uno de ellos.
        hijos.for_all
        (agent (x: PERSONA):BOOLEAN do Result := x.getEdad >= 0 end)
 end
...
invariant
    hijos.for_all
        (agent (x: PERSONA):BOOLEAN do Result := x.getEdad >= 0 end)

end
```

Figura 4.10: Invariantes

El invariante agregado a la clase *Adulto* permite especificar que el valor del atributo *edad* de todos sus hijos sea mayor o igual a cero, durante el ciclo de vida de una instancia de la clase. Hay un tipo de invariante llamado ***loop invariant***, el cual es utilizado dentro de iteraciones garantizando que en cada una de ellas se cumpla una condición.

Cláusula OLD

Para realizar las expresiones que forman parte de las especificaciones de las post-condiciones, Eiffel cuenta con una cláusula muy útil llamada ***old***, que permite evaluar expresiones al comienzo de la ejecución de la rutina que la contenga. La cláusula ***old*** es de la forma ***old*** *EXP*, donde *EXP* es una expresión a evaluar por dicha cláusula. El efecto de la ejecución de dicha expresión es la de evaluar la expresión *EXP* al comienzo de la rutina que contiene dicha expresión.

Eiffel tiene variables especiales, entre ellas encontramos las llamadas variables asociadas, ***VA***, las cuales tienen la función de almacenar el resultado de la evaluación de una expresión, tal es el caso de la expresión ***old EXP***, que se almacena en una variable asociada para su futura utilización.

El funcionamiento normal de la evaluación de la expresión ***old***, ***EO***, es la siguiente:

1. Se evalua EO
2. Si dicha evaluación dispara una excepción, se registra dicho evento en una excepción asociada para su futura evaluación.
3. De lo contrario se asigna EO a VA.
4. Continúa con la expresión que la contenía.

La razón por la cual la posible excepción, resultado de la evaluacion de EO, se almacena en una excepción asociada se explica con el siguiente ejemplo:

```
((old v) /= 0) AND x = y implies old (1/v) = x
```

En este ejemplo se aprecia que la expresión *old(1/v)* puede disparar una excepción de *Aritmetic Overflow* si v = 0, esto solo es posible saberlo al comienzo de la ejecución del método, ya que las expresiones ***old*** se ejecutan en ese momento, pero en caso de que se dispare dicha excepción, ella debe ser retenida y almacenda, ya que si al finalizar la ejecución no se cumple *((old v) /= 0) AND x = y*, entonces la excepción no tendrá sentido, de lo contrario se disparará la excepción almacenada[34].

Una expresión *old (EO)*, de la forma **old EXP** es válida si satisface las siguientes condiciones.

1. Aparece en las post-condiciones de la rutina.
2. No involucra la cláusula Result.
3. Si reemplazamos EO por EXP la post-condición es valida.

Para mostrar una aplicación se utilizará la clase *PERSONA*, como se muestra en la figura 4.11, especificando que al finalizar el método *incrementarEdad* el valor del atributo *edad* es igual al valor que poseía al comienzo de la ejecución del método mas uno.

```
deferred class PERSONA
 feature
   edad:INTEGER;
   ...
 incremetarEdad()
      do
        edad := edad + 1
ensure
           edad = old edad + 1;
      end
   ...
end
```

Figura 4.11: Cláusula OLD

Muchas veces es deseable especificar que solo ciertas variables van a ser modificadas durante la ejecución de un método, esto es posible especificarlo con la cláusula ***only***, definiendola en las post-condiciones de la rutina. Al especificar por ejemplo ***only*** *x,y,z* se esta indicando que las variables *x,y,s* son las únicas que van a ser modificadas. Una forma alternativa de decir que una varible *x* no cambiará su valor es especificar como post-condición:
x = old x

La especificación de una clase depende del contexto donde se este implementando. Como ya se ha visto para especificar pre y post-condiciones de un método de una clase, se utilizan las expresiones ***require*** y ***ensure***. Esto es válido siempre y cuando el método que se desea especificar no sea un método redefinido de una clase que heredea de otra. En este caso se deben utilizar las cláusulas ***require else*** y ***ensure then***, las cuales tienen la misma utilidad que ***require*** y ***ensure*** respectivamente, con la diferecia de que al utilizar ***require else***, se esta realizado un *OR* con las *assertions* definidas en la clase padre, y con ***ensure then*** se hace un *AND*. Como se puede apreciar con ***require else*** se debilitan las *assertions* del padre y con ***ensure then*** se fortalecen las mismas[34].

Capítulo 5

De la teoría a la práctica

En capítulos anteriores se describió el lenguaje de especificación formal JML y el lenguaje de implementación y especificación Eiffel, explicando su sintaxis y semántica. En este capítulo se especifica una *Lista* mediante una estructura dinámica, dado que en la mayoría de los ejemplos encontrados se especificaba la misma de forma estática [38]. El objetivo del capítulo es especificar un ejemplo sencillo teniendo en cuenta las características de cada lenguaje y probar esa especificación en algunas herramientas estudiadas.

Lista

La idea consiste en declarar una referencia a un nodo que será el primero de la lista. Para esta implementación, también tendremos una referencia al último nodo. Cada nodo contendrá el valor del mismo y una referencia al siguiente nodo. La figura 5.1 muestra el diseño de la Lista.

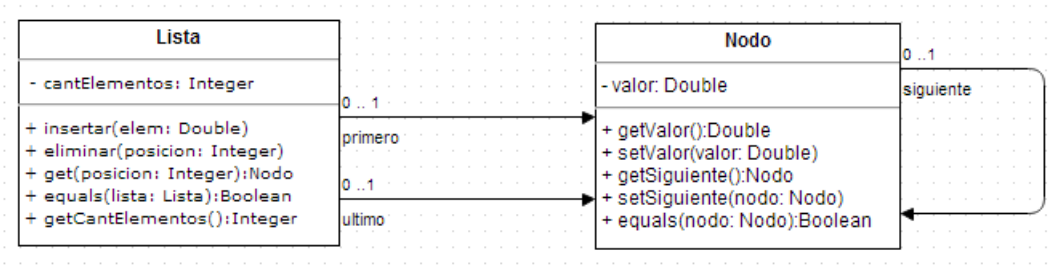

Figura 5.1: Diseño de Lista

Para implementar una Lista es necesario definir la clase *Nodo* y la clase *Lista*. La clase *Nodo* contiene el valor del nodo, en este caso se eligió el valor de tipo *double*, y un puntero al siguiente nodo. En la misma se implementó el constructor, las funciones *getters* y los métodos *setters* para los atributos. A continuación se muestra la especificación informal de dicha clase, la cual no se entrará en detalle dado la simplicidad de la misma.

```
CLASS NODO {

  DOUBLE valor;
  NODO siguiente;

  Post-condición:  El valor del atributo 'valor' es igual
                   al parámetro de entrada 'elem'
```

```
  Post-condición:  El atributo 'siguiente' es inicializado
                   sin referencia
  NODO(DOUBLE elem){ ... }

  Post-condición:  El valor que retorna la función es igual
                   al atributo 'valor'
  DOUBLE getValor(){ ... }

  Post-condición:  El valor del atributo 'valor' es igual al
                   parámetro de entrada 'item'
  VOID setValor(DOUBLE item){ ... }

  Post-condición:  El valor que retorna la función es igual
                   al atributo 'siguiente'
  NODO getSiguiente(){ ... }

  Post-condición:  El atributo 'siguiente' es igual al parámetro
                   de entrada 'sig'
  VOID setSiguiente(NODO sig){ ... }
}
```

La clase *Lista* tiene un atributo, *primero*, y otro *ultimo*, de tipo *Nodo*, que contiene la referencia al primer y último nodo de la lista respectivamente, y un atributo *cantElementos* que indica la cantidad de nodos en la lista.

Se desea realizar una especificación muy detallada, *strong specification*, del método *insertar* de la clase *Lista*, para el cual se define la siguiente post-condición en lenguaje natural:

```
La lista resultado contiene el nodo recibido por parámetro como
primer nodo y el resto de la lista es igual a la lista al momento
de invocar al método.
```

Si se vincula esta post-condición con la implementación del TAD Lista, descrita anteriormente, se pueden desprender las siguientes post-condiciones de la post-condición enunciada anteriormente.

```
Post-condición 1:  El atributo 'cantElementos' se incrementa en 1.
Post-condición 2:  El atributo 'primero' hace referencia a un nuevo nodo
                   con valor 'elem' y el atributo 'ultimo' al último
                   nodo de la lista.
Post-condición 3:  Los nodos de la lista resultado a continuación del
                   primer nodo son exactamente los mismos y están en el
                   mismo orden que los nodos de la lista al momento de
                   invocar al método.
```

Estas post-condiciones se muestran a continuación, junto con la especificación informal de la función *get*.

```
CLASS LISTA {

   NODO primero;
   NODO ultimo;
   INTEGER cantElementos;

   LISTA(){ ... }
   ...

   Pre-condición:    El parámetro de entrada 'posicion' tiene
```

```
                    que ser mayor o igual a 0 y menor que el
                    atributo 'cantElementos'
   Post-condición:  El Nodo que retorna es el que se encuentra en la
                    posición 'posicion' de la lista
   public Nodo get(int posicion){ ... }

   Post-condición 1:  El atributo 'cantElementos' se incrementa en 1.
   Post-condición 2:  El atributo 'primero' hace referencia a un
                      nuevo nodo con valor 'elem' y el atributo
                      'ultimo' al ultimo nodo de la lista.
   Post-condición 3:  Los nodos de la lista resultado a continuación
                      del primer nodo son exactamente los mismos y
                      están en el mismo orden que los nodos de la
                      lista al momento de invocar al método.
   public void insertar(double elem){ ... }
}
```

Para evitar ambigüedad al especificar un contrato de software de manera informal, se refinan con mayor detalle las post-condiciones 2 y 3 respectivamente. La post-condición 2 se puede dividir en las siguientes post-condiciones:

```
Post-condición 2.1: El atributo 'valor' del nodo referenciado por el atributo
                    'primero' es igual al parámetro 'elem'.
Post-condición 2.2: El atributo 'ultimo' hace referencia al nodo de la lista
                    ubicado en la ultima posición (tamaño de la lista al momento
                    de invocar al método) y el atributo 'siguiente' de dicho
                    nodo es 'null'.
```

La post-condición 3 es refinada en una post-condición como se muestra a continuación:

```
Post-condición 3.1: El nodo en la lista resultado ubicado en una cierta
                    posición i, es el mismo que el nodo en la posición i-1
                    de la lista antes de invocar el método. Esto se cumple para
                    todo i entre 1 y el tamaño de la lista resultado menos 1.
```

En las siguientes secciones se muestra la especificación formal del ejercicio planteado para cada lenguaje y los inconvenientes hallados.

5.1. Especificación en JML

En el capítulo 3 se mencionaron y categorizaron las principales herramientas utilizadas en el JML. En esta subsección se especifica formalmente el ejercicio propuesto y se prueba dicha especificación con algunas herramientas. Se decidió utilizar herramientas comprendidas en las categorías **Chequeo en tiempo de ejecución de *assertions*** y **Chequeo estático y verificación**, para estudiar dos tipos de enfoques distintos y complementarios en la especificación formal. De cada una de las categorías se seleccionaron las herramientas que se consideran mas adecuadas, en términos de continuidad, actualizaciones de versiones y disponibilidad de documentación. Las herramientas seleccionadas dentro de la categoría Chequeo en tiempo de ejecución de *assertions* son jml4c y OpenJML, y dentro de la categoría Chequeo estático y verificación se eligió la herramienta Key.

Las herramientas pertenecientes a la primer categoría chequean contratos en tiempo de ejecución. En cambio las herramientas pertenecientes a la categoría *Chequeo estático y verificación* utilizan técnicas lógicas para probar la correctitud, se prueba que no haya violaciones de las especificaciones sin necesidad de ejecutar el código. Esta categoría a menudo provee una mejor verificación, ya que recorre todos los caminos posibles de ejecución y prueba que las especificaciones se cumplan en ellos. Sin embargo estas

ventajas tienen un precio, las herramientas de esta categoría generalmente requieren especificaciones más completas, no solo para el módulo a ser chequeado, sino también para los módulos y librerías de los que depende.

Para ejecutar las herramientas de la categoría Chequeo en tiempo de ejecución de *assertions*, se utiliza una clase en java llamada *Test*, que crea una instancia de la clase *Lista* y le agrega nodos utilizando el método *insertar*.

La herramienta Key puede ser utilizada a través del sitio web de la herramienta, utilizando la tecnología Java *WebStart* o instalando la aplicación de escritorio. Esta herramienta permite cargar las clases *.java*, las cuales son examinadas por un analizador sintáctico y lexicográfico. Si las clases se cargan con éxito significa que la especificación es sintácticamente correcta, permitiendo al usuario elegir un método de una de las clases para ser verificado. Esta verificación se realiza a partir de técnicas lógicas como se mencionó anteriormente. En caso de que las clases no puedan cargarse, se notificará al usuario y la verificación no podrá realizarse.

Se presenta a continuación la implementación en Java y especificación en JML, de las clase *Nodo* mencionada anteriormente. Los atributos *valor* y *siguiente* pueden ser accedidos desde la clase *Lista*, dado que tienen la cláusula *spec_public*. Otra alternativa utilizada, es agregar la cláusula *pure* a las funciones *getters* de los atributos.

```
public class Nodo {

    private /*@ spec_public @*/ double valor;
    private /*@ spec_public @*/ /*@ nullable @*/ Nodo siguiente;

    //@ assignable valor, siguiente;
    //@ ensures this.valor == elem && this.siguiente == null;
    Nodo(double elem){ ... }

    //@ ensures \result == this.valor;
    public /*@ pure @*/ double getValor(){ ... }

    //@ ensures this.valor == elem;
    public void setValor(double elem){ ... }

    //@ ensures \result == this.siguiente;
    public /*@ pure @*/ /*@ nullable @*/ Nodo getSiguiente(){ ... }

    //@ ensures this.siguiente == sig;
    public void setSiguiente(/*@ nullable @*/ Nodo sig){ ... }
}
```

A continuación se estudia en detalle las post-condiciones del método *insertar*, perteneciente a la clase *Lista*.

La post-condición 1 y la post-condición 2.1 mencionadas anteriormente, son bastante sencillas y no presentan inconvenientes. Para especificar la post-condición 1 se utiliza la cláusula *old*, vista en el capítulo 2, para verificar que el atributo *cantElementos* se incrementa en 1. A continuación se muestra la especificación en JML.

```
//@ ensures cantElementos == \old(cantElementos) + 1;
```

En la post-condición 2.1 se verifica que el atributo *primero* no sea *null* y el atributo *valor* del nodo insertado sea igual al parámetro de entrada *elem*. A continuación se presenta la especificación en JML

```
//@ ensures primero != null && primero.valor == elem;
```

Por otro lado, se encontraron varias maneras de especificar la post-condición 2.2 y la post-condición 3.1. Se presentan las opciones encontradas para especificar estas post-condiciones, utilizando distintos elementos del JML.

1. Utilizando la función *get*.
2. Utilizando la cláusula *reach*.
3. Utilizando modelos

A continuación se detalla cada especificación propuesta para la post-condición 2.2 y post-condición 3.1.

Utilizando la función *get*

La post-condición 2.2 se puede dividir en dos partes, por un lado hay que obtener el nodo en la última posición de la lista, tamaño de la lista - 1 y verificar que el atributo *ultimo* haga referencia a ese nodo. Por otro lado hay que verificar que el atributo *siguiente* del nodo en la última posición de la lista sea *null*. La función *get*, perteneciente a la clase *Lista*, permite obtener un nodo en una determinada posición de la lista. Dadas las características del JML, fue imposible especificar la post-condición de la función *get*, sin tener que utilizar la misma función o una función similar, por lo que se decidió especificarla de manera informal. La misma se muestra a continuación.

```
/*@
@ requires posicion<cantElementos;
@ ensures (* El Nodo retornado se encuentra en la
@               posición 'posicion' de la lista *)
public /*@ pure @*/ Nodo get(int posicion){ ... }
```

Para obtener el nodo en la última posición, se debe invocar la función *get* con el índice *cantElementos* - 1, el atributo *siguiente* de este nodo debe ser *null*. Si el atributo *ultimo* hace referencia al último nodo de la lista, entones se verifica que el atributo *siguiente* de *ultimo* es *null*. Por lo tanto, en la especificación en vez de volver a obtener el último elemento de la lista y verificar que sea *null*, solo se va a verificar que el atributo *siguiente* de *ultimo* sea *null*, sin perder generalidad. La especificación de la post-condición 2.2 se presenta a continuación.

```
El atributo 'ultimo' hace referencia al nodo de la lista
ubicado en la ultima posición (tamaño de la lista al momento
de invocar al método) y el atributo 'siguiente' de dicho
nodo es 'null'.

//@ ensures get(cantElementos-1) == ultimo && ultimo.siguiente == null;
```

Para cumplir con la post-condición 3.1 se debe recorrer los nodos de la lista que se encuentren entre 1 y *cantElementos* - 1, dado que la inserción es al principio. Se verifica que el nodo en el índice *j-1* antes de la invocación del método, utilizando la cláusula *old*, sea el mismo en el índice *j* al finalizar la ejecución. Para verificar la post-condición no es suficiente que las referencias a los nodos coincidan, dado que la ejecución del método podría modificar el valor de alguno de los nodos. Por lo tanto, además de verificar que las referencias sean las mismas, se verifica que los valores de los nodos sean iguales. En esta especificación también se utiliza la función *get* para obtener el nodo en una posición determinada. A continuación se muestra dicha especificación.

```
El nodo en la lista resultado ubicado en una cierta
posición i, es el mismo que el nodo en la posición i-1
de la lista antes de invocar el método. Esto se cumple para
todo i entre 1 y el tamaño de la lista resultado menos 1.

/*@
@ ensures (\forall int i; (0 < i) && (i < cantElementos);
@           \old(this.get(i-1)) == this.get(i) &&
@           \old(this.get(i-1).valor) == this.get(i).valor)
@*/
```

La especificación completa del método *insertar* utilizando la función *get* se presenta en la figura 5.2

```
/*@
@ ensures primero != null && primero.valor == elem
@   && cantElementos == \old(cantElementos) + 1
@   && get(cantElementos-1) == ultimo && ultimo.siguiente == null
@   && (\forall int i; (0 < i) && (i < cantElementos);
@          \old(this.get(i-1)) == this.get(i) &&
@          \old(this.get(i-1).valor) == this.get(i).valor);
@*/
public void insertar(double elem){ ... }
```

Figura 5.2: Especificación del método *insertar* utilizando la función *get* en JML

A continuación se mostrará el comportamiento de las herramientas para la especificación de la clase *Lista* y *Nodo*.

jml4c

El código no compiló de forma correcta porque la herramineta no reconoce la variable i utilizada en la segunda expresión *booleana* de la cláusula *forall*. Se probó separar las dos condiciones definidas en el *forall*, por una cláusula *forall* para cada condición, como se muestra a continuación:

```
/*@
@  (\forall int i; (0 < i) && (i < cantElementos);
@      \old(this.get(i-1)) == this.get(i)) &&
@  (\forall int i; (0 < i) && (i < cantElementos);
@      \old(this.get(i-1).valor) == this.get(i).valor);
@*/
```

Con este cambio la herramienta compiló correctamente sin desplegar ningún mensaje de advertencia. Al ejecutar el código usando la clase *Test.java* dió el error *java.lang.NullPointerException* en una línea que no corresponde a ninguna linea de la clase fuente, *Lista*; ya que la misma corresponde a una linea interna del proceso de compilación [37]. No encontramos documentación que detalle este comportamiento del compilador.

OpenJML

Para esta herramienta, las especificaciones públicas y protegidas, *spec_public* y *spec_private*, no funcionan correctamente en tiempo de compilación [31]. Para que funcione en esta herramienta, se dejaron como públicos los atributos *valor* y *siguiente*. Si bien con este cambio compiló correctamente, no se pudo realizar un chequeo en tiempo de ejecución, dado que la misma no implementa el tipo o número de declaraciones para el cuantificador *forall*. El siguiente es el mensaje de advertencia que aparece al momento de la compilación.

```
[exec] WARNING Note: Runtime assertion checking is not implemented for this
               type or number of declarations in a quantified expression
[exec]  @  && (\forall int i; (0 < i) && (i < cantElementos);
[exec]         ^
```

Key

La clase *Nodo* y la clase *Lista* se cargaron correctamente, por lo que las especificaciones eran sintácticamente correctas. Al elegir la clase *insertar*, para realizar la verificación estática, el código cumplió con la especificación para todas pruebas que realizó internamente la herramineta.

Utilizando la cláusula *reach*

Otra alternativa para obtener y recorrer nodos de la lista es utilizar la cláusula *reach*. Este operador, presentado en el capítulo 2, permite hacer referencia a un conjunto de objetos accesibles desde algún objeto en particular. Esta cláusula devuelve el conjunto de nodos pertenecientes a la lista de tipo *JMLObjectSet*. Para cumplir con la post-condición 2.2 se invoca el método *toArray*, perteneciente a la clase *JMLObjectSet*, utilizando el índice *old(reach(primero)).int_size()* para obtener el nodo en la última posición de la lista y se verifica que el atributo *ultimo* haga referencia a este nodo. Notar que en este caso no se utiliza el atributo *cantElementos*. Para verificar que el atributo *siguiente* del último nodo de la lista sea igual a *null* se verifica que el atributo *siguiente* de *ultimo* sea igual a *null*, como se describió anteriormente. A continuación se especifica la *post-condición 2.2*.

```
El atributo 'ultimo' hace referencia al nodo de la lista
ubicado en la ultima posición (tamaño de la lista al momento
de invocar al método) y el atributo 'siguiente' de dicho
nodo es 'null'.

/*@ ensures
@ (\reach(primero).toArray()[\old(\reach(primero)).int_size()]) == ultimo
@  && ultimo.siguiente == null;
@*/
```

Al igual que en la post-condición 2.2, en la post-condición 3.1 se utiliza el método *toArray* para obtener los nodos de la lista y así poder comparar los nodos de los mismos. Existen otros métodos en *JMLObjectSet* tales como *has* (verifica que el nodo pertenezca al conjunto), *isSubset* (verifica si un conjunto es subconjunto del otro), e *intersection* (devuelve la intersección de ambos conjuntos), estos métodos solo permiten especificar la pertenencia de un nodo en un conjunto pero no sirven para comprobar el orden. Dado que la inserción en la lista es al inicio se verifica que el nodo, referencia y valor del mismo, en el índice *j-1* antes de la invocación del método, se encuentre en el índice *j* luego de la misma. A continuación se muestra la especificación.

```
El nodo en la lista resultado ubicado en una cierta
posición i, es el mismo que el nodo en la posición i-1
de la lista antes de invocar el método. Esto se cumple para
todo i entre 1 y el tamaño de la lista resultado menos 1.

/*@ ensures
@(\forall int j; 0 < j && j <= \old(\reach(primero)).int_size();
@    (\old(\reach(primero)).toArray()[j-1]) ==
@       (\reach(primero).toArray()[j]) &&
@     (\old(\reach(primero)).toArray()[j-1].valor) ==
@       (\reach(primero).toArray()[j].valor));
@*/
```

La especificación completa del método *insertar* utilizando la cláusula *reach* se presenta en la figura 5.3

```
/*@
@ ensures primero != null && primero.valor == elem
@   && cantElementos == \old(cantElementos) + 1
@   && (\reach(primero).toArray()[\old(\reach(primero)).int_size()]) == ultimo
@   && ultimo.siguiente == null
@   && (\forall int j; 0 < j && j <= \old(\reach(primero)).int_size();
@        (\old(\reach(primero)).toArray()[j-1]) ==
@         (\reach(primero).toArray()[j]) &&
@        (((Nodo)\old(\reach(primero)).toArray()[j-1]).valor) ==
@         ((Nodo)\reach(primero).toArray()[j]).valor));
@*/
public void insertar(double elem){ ... }
```

Figura 5.3: Especificación del método *insertar* utilizando la cláusula *reach* en JML

A continuación se muestra el comportamiento de las herramientas frente a estas especificaciones.

jml4c

Esta especificación no compiló correctamente, se desplegaron los siguientes mensajes de error:

```
[exec] ERROR Lista.java (at line 31)
[exec]  @  && (\forall int j; 0 < j && j <= \old(\reach(primero)).int_size();
[exec]                                              ^^^^^^
[exec] The method \reach(Nodo) is undefined for the type Lista
[exec] ----------
[exec] ERROR in Lista.java (at line 32)
[exec]  @    (\old(\reach(primero)).toArray()[j-1])
[exec]              ^^^^^^
[exec] The method \reach(Nodo) is undefined for the type Lista
[exec] ----------
[exec] ERROR in Lista.java (at line 33)
[exec]  @     \reach(primero).toArray()[j];
[exec]         ^^^^^^
[exec] The method \reach(Nodo) is undefined for the type Lista
```

Esto significa que la cláusula *reach* no es reconocida por el compilador, dado que no está implementado en jml4c.

OpenJML

En esta herramienta, la especificación compiló correctamente pero se desplegó el siguiente mensaje de advertencia:

```
[exec] WARNING Lista.java:31: Note: Not implemented for runtime assertion checking:
                          ensures clause containing \reach
[exec]  @  && (\forall int j; 0 < j && j <= \old(\reach(primero)).int_size();
[exec]                                                  ^
```

Para esta herramienta, la cláusula *reach* es reconocida por el compilador, pero no está implementada para el chequeo en tiempo de ejecución. En este caso al probar la especificación con la clase *Test.java*, los nodos se agregan de forma correcta. Dado este resultado, se probó la misma especificación pero verificando que el nodo en el índice j antes de la invocación sea igual al nodo en el índice j al terminar la misma, como se muestra a continuación:

```
/*@ ensures
@(\forall int j; 0 < j && j <= \old(\reach(primero)).int_size();
@    (\old(\reach(primero)).toArray()[j]) ==
@      \reach(primero).toArray()[j] &&
@    (((Nodo)\old(\reach(primero)).toArray()[j]).valor) ==
@      ((Nodo)\reach(primero).toArray()[j]).valor);
@*/
```

Esta especificación no es verdadera, dado que la inserción es al comienzo de la *Lista* y los nodos no pueden coincidir en el mismo índice antes y al finalizar la ejecución, sin embargo los nodos se agregaron a la *Lista*. Podemos concluir que la especificación con la cláusula *reach* no es tomada en cuenta por el compilador en tiempo de ejecución.

Key

Al cargar las clases *Nodo* y *Lista*, la herramienta encontró errores sintácticos en la especificación, por lo que estas clases no pudieron ser cargadas de forma exitosa. Frente a este inconveniente con la herramienta, se hizo contacto con un miembro del soporte de *Key* y en pocos días obtuvimos la respuesta de *Daniel Bruns*, que afirmó que los modelos tales como la clase *JMLObjectSet* no son compatibles con *Key*, y que la sintáxis de esta cláusula es diferente a la que se describió en el capítulo 2.

A continuación se muestra la post-condición 2.2 y 3.1 para *key* utilizando la sintáxis adecuada para esta herramienta. Con este cambio las clases se cargaron correctamente, pero para el método *insertar* la herramienta no pudo demostrar todas las pruebas que define a partir de las especificaciones. Luego de utilizar la herramienta con varios ejemplos se observa que la misma realiza pruebas muy completas, las cuales pueden no ser demostradas a causa de una especificación parcial o por una definición inadecuada de los métodos. Se verificó, a nuestro criterio, la correcta y completa especificación del método *insertar*, así como también los métodos que son invocados en dicha implementación y especificación, pero no pudo ser posible lograr que la herramienta verifique la totalidad de las pruebas creadas para dicho método. Dado que el tiempo era acotado y que el manejo de esta herramienta no formaba parte del objetivo del proyecto, no se continuó investigando este punto.

```
Post-condicion 2.2: El atributo 'ultimo' hace referencia al
nodo de la lista ubicado en la ultima posición (tamaño de la
lista al momento de invocar al método) y el atributo 'siguiente'
de dicho nodo es 'null'.
```

```
Post-condicion 3.1: El nodo en la lista resultado ubicado en una
cierta posición i, es el mismo que el nodo en la posición i-1
de la lista antes de invocar el método. Esto se cumple para
todo i entre 1 y el tamaño de la lista resultado menos 1.
```

```
/*@
@ \reach(primero.siguiente,primero,ultimo,cantElementos-1)
@  && ultimo.siguiente == null
@  && (\forall Nodo e; (\forall int j; 0 <j;
@       \old(\reach(primero.siguiente,primero,e,j-1))
@          ==> \reach(primero.siguiente,primero,e,j)));
@*/
```

Utilizando modelos

Otra forma de implementar listas es utilizando modelos, por ejemplo *JMLObjectSequence*, que ya tiene métodos como *insertFront*, que inserta el primer nodo en la lista; pero como se mencionó en el capítulo 2, un modelo se utiliza para especificar variables y métodos en JML, los cuales solo podrán ser utilizados en *assertions*. Esto implica que se deberá tener por un lado una lista para la especificación y

por otro lado una lista que se utilice en la implementación. Por esta razón, la post-condición 3 queda definida de la siguiente manera: Los valores de los nodos de la lista luego de la ejecución, a partir del segundo nodo, son iguales y están en el mismo orden que los valores de los nodos de la lista modelo a partir del segundo nodo. Por lo tanto la post-condición 3.1 queda de la siguiente manera: Para cada indice i entre 1 y *elementos.int_size()* (tamaño de la lista modelo) se obtienen los nodos de la lista modelo en dicha posición y se comparan los mismos con los nodos de la lista resultado ubicados en la misma posición. Para obtener un nodo de la lista resultado en una cierta posición se utiliza la función *get*, la cual se usa para especificar la post-condición 2.2, presentada anteriormente. Se muestra en la figura 5.4 la especificación del método *insertar* utilizando modelos.

```
//@ public ghost JMLObjectSequence elementos;

public Lista(){
   //@ set elementos = new JMLObjectSequence();
   ...
}

/*@
@ assignable elementos;
@ ensures primero != null && primero.valor == elem
@   && cantElementos == \old(cantElementos) + 1
@   && get(cantElementos-1) == ultimo && ultimo.siguiente == null
@   && (\forall int j; 0 < j && j < elementos.int_size();
@          ((Nodo)elementos.itemAt(j)).valor == get(j).valor);
@*/
public void insertar(double elem){
    //@ set elementos = elementos.insertFront(new Nodo(elem));
    Nodo nodo = new Nodo(elem);
    nodo.setSiguiente(primero);
    ...
}
```

Figura 5.4: Especificación del método *insertar* utilizando *modelos* en JML

A continuación se muestra el comportamiento de las herramientas pertenecientes a la categoría Chequeo en tiempo de ejecución de *assertions*, dado que los modelos no son compatibles con *Key*.

jml4c

Esta especificación no compiló correctamente, se desplegaron los siguientes mensajes de error:

```
[exec]  //@ set elementos = new JMLObjectSequence();
[exec]           ^^^^^^^^^
[exec] elementos cannot be resolved
```

Esto significa que en esta herramienta no se reconoce la variable *elementos* dentro del constructor y del método *insertar*.

OpenJML

En esta herramienta la especificación compiló y ejecutó de forma correcta.

En conclusión, se encontró que las herramientas jml4c y OpenJML, pertenecientes a la categoría Chequeo en tiempo de ejecución de *assertions*, no soportaron las dos primeras alternativas propuestas

para especificar el método *insertar*. Se pudo probar con la herramienta OpenJML la utilización de modelos. Para el caso de la herramienta Key, perteneciente a la categoría Chequeo estático y verificación, funcionó correctamente utilizando la función *get*, pero no se verificaron todas las pruebas utilizando la cláusula *reach*. Tampoco funcionó utilizando modelos, dado que la misma no los soporta.

5.2. Especificación en Eiffel

En esta sección se presenta la implementación y especificación de las clases involucradas en el desarrollo del tipo abstracto *Lista*, utilizando Eiffel como lenguaje de implementación y especificación formal. Además, se detallarán los problemas y soluciones encontradas, en particular para la clase *Lista*, se presenta la implementación y especificación del método *insertar*. Mediante este problema se busca experimentar la especificación de elementos de software que involucren punteros en su implementación, utilizando el lenguaje Eiffel.

Dado que la solución de la clase *Nodo* no presenta características interesantes a mencionar, no se discutirá la misma y solo se presentará su solución. A continuación se muestra la especificación de dicha clase utilizando el lenguaje Eiffel.

```
class
   NODO
feature
   valor: DOUBLE
   siguiente: NODO

   setValor(n:DOUBLE)
    do
      valor := n
      ensure
        Current.valor = n
    end

   getValor():DOUBLE
    do
      Result := valor
      ensure
        Result = Current.valor
    end

   setSiguiente(n: NODO)
    do
      siguiente := n
      ensure
        Current.siguiente = n
    end

   getSiguiente():NODO
    do
      Result := siguiente
      ensure
        Result = siguiente
    end
end
```

Eiffel posee una clase, llamada *LINKED_LIST*, que implementa el tipo abstracto Lista deseado. En la pagina oficial del lenguaje se encontró la implementación del año 2007 de dicha clase[30], en la

cual se observa que algunos de sus métodos se encuentran parcialmente especificados. En particular el método *put_front*, cuyo comportamiento es el deseado para el método *insertar* a implementar, no posee especificación. La clase *LINKED_LIST* provee métodos que permiten cumplir con los requerimientos del ejercicio, por lo que dicha clase puede ser una solución del mismo.

En lugar de utilizar dicha clase como solución se desea resolver el problema implementando, al igual que con Java y JML, una lista enlazada con los métodos y atributos descritos en la sección Lista de este capítulo.

Se discutirá la solución correspondiente a la clase *Lista* para el método *insertar* el cual, al igual que JML, su desarrollo presentó diferentes inconvenientes.

Para especificar la post-condición 1 de este método solo es necesario comparar el valor del atributo *cantElementos* de la clase con el valor de dicho atributo al comienzo de la ejecución mas uno. Para especificar la post-condición 2.1 se verifica que el valor del atributo *primero* no sea *Void* (referencia nula) y luego se compara con el valor del atributo *valor* del nodo referenciado por el atributo *primero* para verificar su igualdad. La post-condición 2.2 compara la referencia del atributo *ultimo* con el último nodo de la lista, que se encuentra en la posición *cantelementos* -1, y se verifica que el atributo *siguiente* del nodo tenga valor *Void*. Para obtener el nodo en dicha posición se utiliza la función *get*, que permite obtener el elemento en una cierta posicición, mediante un índice recibido por parámetro. A continuación se muestran las especificaciones mencionadas anteriormente.

```
Post-condición 1:   El atributo 'cantElementos' se incrementa en 1.
Post-condición 2.1: El atributo 'valor' del nodo referenciado por el atributo
                    'primero' es igual al parámetro 'elem'.
Post-condición 2.2: El atributo 'ultimo' hace referencia al
                    nodo de la lista ubicado en la ultima posición (tamaño de la
                    lista al momento de invocar al método) y el atributo 'siguiente'
                    de dicho nodo es 'null'.

ensure
   cantElementos = old cantElementos + 1;
   primero /= Void AND primero.getvalor = elem;
   get(cantelementos-1) = ultimo AND ultimo.getsiguiente = Void;
```

Para especificar la post-condición 3.1 se decidió iterar la lista para poder chequear que el estado de la misma, luego de la ejecución de la función, fuese el esperado. Para ello se debe recorrer la lista resultado comenzando desde su segundo nodo, comparando uno a uno los nodos de la misma con los nodos que poseía al comienzo de la ejecución de la función. La lista a desarrollar no implementa alguna interfaz de listas proporcionada por Eiffel y ademas no extiende alguna de las implementaciones que posee Eiffel de listas, a causa de ello, no es posible utilizar la función *for_all*, al ser una función propia de dichas clases. Eiffel no posee una función similar a *forAll*, provista por el JML, la cual permite iterar un conjunto de objetos definidos que cumplen una cierta condición. Se encontró entonces la primer dificultad al carecer de una herramienta propia de Eiffel que permita iterar la lista, sin necesidad de recorrer la misma con una función auxiliar. En este punto se encuentran dos caminos para la resolución del ejercicio:

1. La clase *Lista* debe extender alguna implementación del TAD lista enlazada provista por Eiffel y de ese modo poder utilizar la función for_all.
2. Se debe buscar un modo alternativo para recorrer la lista.

Si se opta por el primer punto, una posible solución al ejercicio planteado puede ser extender la clase *LINKED_LIST* y sobreescribir los métodos que se consideren necesarios, esto implicaría utilizar la estructura de la clase *Lista* desarrollada por Eiffel, con sus atributos y métodos. Dicha implementación dista considerablemente de la Lista propuesta por el ejercicio. Dado que el objetivo de este ejercicio es implementar la lista descrita en la sección Lista y con ello experimentar la especificación de componentes que involucren punteros, se decidió implementar la clase *Lista* sin extender alguna implementación de lista provista por Eiffel. De ese modo se busca concentrarse en experimentar la especificación de la Lista sin entrar en la complejidad de extender la clase *LINKED_LIST* y sobreescribir sus métodos.

Al optar por el segundo punto se encontró una forma indirecta de recorrer la lista, la misma consiste en recorrer los enteros comprendidos entre el valor cero y el valor *cantelementos* menos uno. Por cada nodo iterado se utiliza la función *get* para obtener el nodo en la lista en dicha posición. De esta forma cada nodo obtenido de la lista resultado es comparado con el nodo en una posición anterior en la lista considerando su estado al comienzo de la ejecución de la función. A continuación se muestra la especificación de la post-condición 3.1 del método *insertar*.

```
El nodo en la lista resultado ubicado en una cierta
posición i, es el mismo que el nodo en la posición i-1
de la lista antes de invocar el método. Esto se cumple para
todo i entre 1 y el tamaño de la lista resultado menos 1.

ensure
   (1 |..| (cantElementos-1)).for_all
   (agent (x: INTEGER):BOOLEAN do
   Result := (old Current.get(x-1) = Current.get(x)) end);
```

Al momento de compilar dicha solución se encontró con el inconveniente de que la expresión *old* proporcionada por Eiffel no puede ser utilizada en un *scope* distinto al de la post-condición. En este caso la expresión *old* es utilizada en la post-condición pero dentro de la función *agent*, por lo que el *scope* considerado para dicha expresión es el *scope* de la función *agent*. Es por ello que esta especificación no es válida, ya que para que la comparación sea correcta es necesario comparar los nodos de la lista utilizando la función *old* dentro de la función *agent*. Este punto deja claro que aunque la lista solución tenga las funcionalidades de una lista de Eiffel, este método de iteración no puede ser empleado ya que es necesario, en todos los casos, hacer referencia a los nodos de la lista al comienzo de la ejecución del método.

Dado que no se encontró una solución al problema, debido a que el lenguaje no proporciona las herramientas necesarias para recorrer la lista, y al mismo tiempo hacer referencia a la misma al comienzo de la ejecución del método, se optó por especificar la clase *Lista* utilizando funciones auxiliares. De este modo se desea saber que elementos, no proporcionados por Eiffel, son necesarios para poder especificar completamente dicha clase.

Para poder especificar la lista es necesario obtener su estado al comienzo de la ejecución del método *insertar*, para asi poder comparar sus nodos con los nodos que posee la misma al finalizar la ejecución mismo. Es necesario entonces contar con alguna sentencia o conjunto de ellas que permita obtener dicha lista.

La cláusula *Current* utilizada dentro de la clase *Lista* es la referencia a la instancia actual de dicha clase, por lo que el resultado de ejecutar la sentencia *old Current* en la post-condición retornará el valor *Current*, ya que la referencia a la instancia de la clase *Lista* no varía luego de cada inserción.

Si se utiliza la sentencia *old Current.twin* el resultado será una nueva instancia de la clase *Lista* la cual contiene las mismas referencias que la instancia original, por lo que contendrá los mismos nodos que la lista original. Esta lista podría ser utilizada para comparar los elementos de la lista resultado a continuación del primer elemento, pero una mala implementación del método *insertar* podría agregar, por ejemplo, además del primer nodo, otro nodo en la lista. Por lo tanto este nuevo nodo esta presente en la lista original y en la lista retornada por la sentencia *old Current.twin*, lo que causaría que este nuevo nodo no sea detectado. Esto se produce ya que no se genera una copia total de la lista, solo se realiza una copia de la instancia de la clase *Lista* y no de los nodos que hace referencia.

Al utilizar la sentencia *old Current.deep_twin* se genera una copia de la instancia de la clase *Lista* y de todos los nodos que referencia de forma recursiva. Dado que la lista resultado es una copia, las referencias a los nodos cambian, por ello la forma de comparación de los nodos de la lista original y esta nueva lista debe ser por el valor que contienen y no por la referencia a los mismos. Por ello se decidió cambiar la post-condición 3.1, para comparar los valores de los nodos en lugar de las referencias, por lo que la post-condición 3.1 queda definida de la siguiente manera:

```
El valor del nodo de la lista resultado ubicado en una cierta posición i, es el mismo
que el valor del nodo en la posición i-1 de la lista antes de invocar el método.
Esto se cumple para todo i entre 1 y el tamaño de la lista resultado menos 1.
```

Se creó un método auxiliar, llamado *forAllReferencia*, capaz de recorrer cada nodo de la lista. Como Eiffel no permite la ejecución de la expresión *old* fuera del *scope* de las post-condiciones, no es posible utilizar dicha expresión en el cuerpo del método a crear, fue así que se decidió que la función reciba por parámetro una lista que será utilizada como lista de referencia. De este modo, cada nodo de la lista que se itera puede ser comparado con los nodos de la lista de referencia. En este caso, la lista que se pasa por parámetro al método *forAllReferencia* es una copia de la lista, realizada al comienzo de la ejecución de la función.

Para permitir que la comparación se realice en un determinado rango de indices, fue necesario que el método *forAllReferencia* reciba por parámetro los indices donde realizar la iteración. Para poder dejar libre la implementación de la función de comparación de los nodos de las listas es necesario agregar un parámetro mas de tipo *agent*, al igual que la expresión *for_all* de Eiffel, la cual también contiene dicho parámetro. El agente recibe por parámetro dos nodos de la lista y retorna un booleano, resultado de su comparación. Además de los parámetros indices que indican desde donde y hasta donde se desea realizar la comparación, se necesita agregar un parámetro mas que indique el *offset* que tendrá la lista de referencia al momento de realizar la comparación. El cabezal del método *forAllReferencia* creado es el siguiente:

```
forAllReferencia (desde, hasta, offsetRef:INTEGER;
f: FUNCTION [ANY, TUPLE[NODO], BOOLEAN];
referencia:LISTA): BOOLEAN
```

La función **f** es la implementación de la función *agent* utilizada para comparar los valores de los nodos, *referencia* es la lista que se utiliza como referencia para la comparación, *desde* y *hasta* son rangos de indices y por último *offsetRef* es el *offset* que tendrá la lista de referencia. Para evitar introducir errores al momento de iterar la lista, en su implementación, el método *forAllReferencia* utiliza la expresión *for_all* para realizar la iteración. Para comparar los valores de los nodos se utiliza la función *getvalor*, perteneciente a la clase *Nodo*. A continuación se muestra la post-condición 3.1 del método *insertar*.

```
El valor del nodo de la lista resultado ubicado en una
cierta posición i, es el mismo que el valor del nodo en
la posición i-1 de la lista antes de invocar el método.
Esto se cumple para todo i entre 1 y el tamaño de la lista
resultado menos 1.

ensure
   Current.forAllReferencia (1,cantElementos-1,-1,
   agent (x1,x2:NODO):BOOLEAN do Result:=x1.getvalor = x2.getvalor end,
   old Current.deep_twin);
```

Utilizando el método *forAllReferencia* fue posible especificar toda la lista como se requería, pero no se logró especificar el método *forAllReferencia* creado y la función *get*. Los mismos no pueden ser especificados dadas las herramientas proporcionadas por el lenguaje y se consideran como funciones de bajo nivel, por lo que no fueron especificadas ya que el código utilizado para especificar dichas funciones sería similar al utilizado para implementar la función que se desea especificar. En la figura 5.5 se presenta la especificación e implementación de la función *insertar* utilizando la función *forAllReferencia* en Eiffel.

```
insertar(elem: DOUBLE)
  local
  aux: NODO;
   do
    create aux;
    aux.setvalor(elem);
    aux.setsiguiente(primero);
    primero := aux;
    if(ultimo = Void) then
      ultimo:=aux;
    end
    cantElementos := cantElementos + 1;
    ensure
      cantElementos = old cantElementos + 1;
      primero /= Void AND primero.getvalor = elem;
      get(cantelementos-1) = ultimo;
      ultimo.getsiguiente = Void;
      Current.forAllReferencia (1,cantElementos-1,-1,
      agent (x1,x2:NODO):BOOLEAN do Result:=x1.getvalor = x2.getvalor end,
      old Current.deep_twin);
   end
```

Figura 5.5: Especificación e implementación del método *insertar* en Eiffel

Los contratos de software involucrados en los métodos de la solución del ejercicio, pudieron ser especificados por completo a diferencia de los métodos *get* y *forAllReferencia*, los cuales no pudieron ser especificados dadas las herramientas provistas por el lenguaje. En términos generales Eiffel cubrió nuestras expectativas al brindar las herramientas necesarias para implementar y especificar el ejercicios propuesto. Si bien en algunos casos requirió de funciones auxiliares, el objetivo se cumplió.

Capítulo 6

Conclusiones y Trabajos a Futuro

En este proyecto se estudió la metodología de diseño por contratos utilizando el JML como lenguaje formal de especificación. Se estudió dicho lenguaje y se investigó el estado actual de algunas herramientas disponibles para su interpretación, compilación y ejecución de pruebas formales. Dado que Eiffel es el lenguaje formal que introdujo la noción de diseño por contrato, se decidió incorporar el estudio del mismo a los objetivos del proyecto de grado, abarcando la implementación y especificación formal, así como también el entorno de desarrollo. A continuación se presentan las conclusiones y trabajo a futuro del estudio realizado.

6.1. Conclusiones

En este proyecto de grado se estudió el enfoque de diseño por contratos (DBC) que se basa en los métodos formales para el diseño e implementación de aplicaciones y componentes. Se comprendió la importancia de los mismos para la construcción de productos de software de calidad, dado que en general no se utilizan estas prácticas en el proceso de desarrollo de software. Si bien los contratos de software pueden escribirse de manera informal, estos pueden llegar a tener varias interpretaciones. Por esta razón fue interesante enfocarnos en el estudio de dos lenguajes formales de especificación, Eiffel y JML.

Ambos lenguajes permitieron especificar contratos de software sencillos, sin embargo resultó complicado especificar algunos contratos mas complejos, por ejemplo los que involucran estructuras de datos dinámicos.

Al estudiar el JML se comprobó que la semántica de algunos elementos o expresiones es todavía objeto de investigación. Esto es consecuencia de que el lenguaje está en etapa de desarrollo. También se constató que la sintaxis y semántica pueden variar dependiendo de la herramienta utilizada.

Se realizó un estudio del estado de varias herramientas y se encontró que la mayoría están en una fase inicial de desarrollo, presentando dificultad al implementar algunos comportamientos del JML. En la práctica pudimos observar que algunas expresiones que son válidas en una herramienta no eran reconocidas en otras.

Con respecto a las herramientas comprendidas en la categoría **Chequeo en tiempo de ejecución de *assertions***, se pudo constatar que en las que fueron examinadas, la mayoría de los errores desplegados tanto en tiempo de compilación como de ejecución, son internos a la implementación de la herramienta y no se explica realmente el motivo del error. Además, como se estudió en el capítulo anterior, las herramientas estudiadas en esta categoría no permitieron compilar y/o ejecutar la mayoría de las especificación de la post-condición del método *insertar*, de la clase *Lista*, que establece que todos los elementos de la lista antes de ejecutar el método, son los mismos y tienen el mismo orden que la lista luego de la invocación, a partir de su segundo elemento.

Para el caso de *Key*, perteneciente a la categoría **Chequeo estático y verificación**, no sigue tal cual la sintaxis y semántica definida en el JML, por ejemplo por no soportar modelos. En esta herramienta

también se pudo especificar el problema planteado en el capítulo anterior.En *Key* las especificaciones escritas en JML se transforman en teoremas de lógica dinámica y luego se comparan contra la semántica del programa, que también están definidos en términos de esta lógica. En esto no se profundizó, ya que el manejo detallado de esta herramienta no formaba parte de los objetivos del proyecto.

Eiffel, a diferencia de JML, incluye comportamientos que permiten especificar elementos de software, por lo que la especificación utilizando dicho lenguaje resulta natural al no requerir una extensión del mismo como sucede con Java y JML. EiffelStudio es el ambiente de desarrollo que utilizamos para Eiffel, el cual brinda mucha más información sobre los errores cometidos durante la implementación y especificación de rutinas o clases, a diferencia de algunas herramientas mencionadas anteriormente. Este lenguaje ofrece, al igual que el JML, una gran variedad de elementos para su especificación, los cuales poseen restricciones en su uso, como se observó en la especificación de la clase *Lista*.

En lo personal este proyecto nos dio una visión diferente con respecto al proceso de desarrollo de software, nos motiva a incorporar el uso de los contratos de software en este proceso para desarrollar software con la menor cantidad de defectos, más allá de los problemas presentados con los lenguajes (jml y eiffel) y con las herramientas estudiadas.

Con respecto a los lenguajes de especificación formal, lo utilizaríamos para especificar aquellos contratos de software que implican el desarrollo de productos altamente críticos. Esto se debe a que muchas veces especificar un contrato de software no resulta sencillo y consume recursos de tiempo, ya sea en su diseño como en su implementación, pudiendo generar riesgo de costo y de tiempo en el proyecto.

6.2. Trabajos a Futuro

En esta sección se mencionan los trabajos a futuros.

Con respecto a nuestro proyecto consideramos que sería interesante continuar con el estudio de la herramienta Key, por un lado para comprender en profundidad el manejo de esta herramienta y por otro lado para realizar especificaciones más complejas, dado que en la práctica fue la herramienta que permitió verificar las especificaciones presentadas en el capítulo anterior.

El proyecto de grado se enfocó en el estudio de herramientas comprendidas en las categorías **Chequeo en tiempo de ejecución de *assertions*** e invariantes y **Chequeo estático y verificación**, dejando afuera herramientas que nos parecen interesantes como *DAIKON* y *HOUDINI*, que sugieren posibles *assertions* a partir de clases implementadas en Java. Estas herramientas pertenecen a la categoría *Generación de especificaciones*. Resulta interesante examinar las mismas y estudiar la posibilidad de obtener buenos candidatos a ser *assertions* del código a especificar, reduciendo el costo de especificación.

Otra categoría estudiada que resultó muy interesante es *Generación y ejecución de casos de pruebas*. En ella se encuentra la herramienta *JMLUNIT*, que genera casos de prueba a partir de especificaciones en JML, pero no se profundizó en ella. La utilización de esta herramienta podría reducir considerablemente el costo de desarrollo de casos de pruebas para verificar dinámicamente los elementos de software especificados con el *JML*.

Con respecto a otros trabajos que se están realizando, nos pareció interesante el desarrollo de un entorno amigable para el JML, dado que resulta complicado localizar un error sintáctico en el lenguaje. Normalmente es utilizado con herramientas como *Eclipse*, *EditPlus*, entre otras, las cuales facilitan la tarea de compilación y ejecución. Por esa razón se quiere desarrollar desde hace un tiempo un IDE para el JML [39].

Otra área de trabajo futuro para JML es la concurrencia. Existen investigaciones donde se agregaron cláusulas a JML para soportar multi-hilo en programas Java [36], de todas formas esto se encuentra en fase experimental.

El auge de los dispositivos móviles hace que actualmente sea necesario especificar el comportamiento de estas aplicaciones. Es deseable que la migración de aplicaciones web y de escritorio a aplicaciones móviles, también incluya la migración de las especificaciones en caso de que las mismas estén presentes. Es por ello que como trabajo a futuro resulta interesante realizar una investigación sobre la posibilidad de especificar aplicaciones móviles.

Bibliografía

[1] *SWEBOK executive editors, Alain Abran, James W. Moore; editors, Pierre Bourque, Robert Dupuis. (2004)* 1

[2] CES *Jeannette M. Wing - A Specifier's Introduction to Formal Methods. 1990 CMU-CS-136* 1, 1.1

[3] Bertrand Meyer. Eiffel *The Language. ObjectOriented Series. Prentice Hall, New York, NY* 1992. 2

[4] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joeseph R. Kiniry, Gary T. Leavens, K. Rustan M. Leino, and Erik Poll *An overview of JML tools and applications. International Journal on Software Tools for Technology Transfer (STTT)* 7(3):212232, June 2005 Random House, N.Y. 2

[5] Gary T. Leavens, Yoonsik Cheon, Curtis Clifton, Clyde Ruby, and David R. Cok. *How the design of JML accommodates both runtime assertion checking and formal veri?cation. Science of Computer Programming* 55(13):185208, March 2005. 2

[6] Gary T. Leavens, Eric Poll, Curtis Clifton, Yoonsik Cheon, Clyde Rudy *JML Referente Manual (revision 1.65)* 28.01.2004. 2.1.4

[7] Cliff B. Jones *Systematic Software Development Using VDM. International Series in Computer Science. Prentice Hall, Englewood Cliffs, N.J., second edition* 1990. 2

[8] K. Rustan M. Leino, Greg Nelson, and James B. Saxe *ESC/Java Userś Manual. SRC Technical Note 2000-02* October, 2000. 2.2.2, 2.2.2

[9] Gary T. Leavens *http://www.jmlspecs.org/* 2.5

[10] Mellema, Gregory, formal language (en inglÃ©s), The Oxford Companion to Philosophy, Oxford University Press, consultado el 13 de octubre de 2009

[11] Joseph R. Kiniry1, Daniel M. Zimmerman2, and Ralph Hyland3 *Testing Library Specifcations by Verifying Conformance Tests.*

[12] Patrice Chalin, Joseph R. Kiniry, Gary T. Leavens, Erik Poll *Beyond Assertions: Advanced Specification and Verification with JML and ESC/Java2.* 1.3, 3, 3.1, 3.2.4

[13] Lilian Burdy, Yoonsik Cheon, David R. Cok, Michael D. Ernst, Joseph R. Kiniry, Gary T.Leavens, K. Rustan M. Leino7, Erik Poll *An overview of JML tools and applications.* 1.3, 3, 3.1, 3.2.1, 3.2.2, 3.2.4, 3.2.4, 3.2.5, 3.2.5, 3.2.6

[14] *http://kindsoftware.com/products/opensource/ESCJava2* 3.2.4

[15] *http://sourceforge.net/apps/trac/jmlspecs/wiki/OpenJml* 3.2.1, 3.2.2

[16] *http://www.cs.utep.edu/cheon/download/jml4c/index.php* 3.2.1

[17] *http://formalmethods.insttech.washington.edu/software/jmlunitng/* 3.2.3

[18] *http://citeseer.ist.psu.edu/viewdoc/summary?doi=10.1.1.16.1895*

[19] Joachim van der Berg, Bart Jacobs *The LOOP compiler from Java and JML* 3.2.4

[20] *http://www-sop.inria.fr/everest/soft/Jack/jack.html* 3.2.4

[21] Claude Marché *The Krakatoa Verification Tool for JAVA programs* 3.2.4

[22] *http://key-project.org/* 3.2.4

[23] Wolfgang Ahrendt, Thomas Baar, Bernhard Beckert, Richard Bubel, Martin Giese, Reiner Hahnle,Wolfram Menzel, Wojciech Mostowski, Andreas Roth, Steen Schlager, Peter H. Schmitt *The KeY Tool* 3.2.4

[24] *Daikon Invariant Detector User Manual* 3.2.5

[25] Michael D. Ernst, Jeff H.Perkins, Philip J. Guo, Stephen McCamant, Carlos Pacheco, Matthew S. Tschantz, Chen Xiao *The Daikon system for dynamic detection of likely invariants* 3.2.5

[26] *http://autojml.sourceforge.net/* 3.2.5

[27] *http://www.cin.ufpe.br/~hemr/JMLAOP/ajmlc.htm* 3.2.1

[28] *http://www.dc.uba.ar/inv/grupos/rfm_folder/TACO* 3.2.4

[29] *http://sdg.csail.mit.edu/forge/jmlforge.html* 3.2.4

[30] http://archive.eiffel.com/products/base/classes/list/linked_list.html 5.2

[31] *David Cok http://openjml.org/* 5.1

[32] http://www.eiffel.com/developers/presentations/ 4.1

[33] http://www.eiffel.com/developers/design_by_contract.html 4.1

[34] Eiffel: Analysis, Design and Programming Language Standard ECMA367 segunda edicion Junio 2006 4.2.1, 4.2.2, 4.2.3, 4.2.3, 4.2.3, 4.2.4, 4.2.5, 4.2.6, 4.2.8, 4.2.8, 4.2.8, 4.2.10, 4.2.11, 4.2.13, 4.2.14, 4.2.14

[35] An Eiffel Tutorial, ISE Technical Report TREI66TU. 4.1

[36] Edwin Rodriguez, Matthew Dwyer, Cormac Flanagan, John Hatcliff, Gary T. Leavens, Robby *Extending JML for Modular Specification and Verification of Multi-threaded Programs* 2005 6.2

[37] Design by Contract with JML, Gary T. Leavens, Yoonsik Cheon, Setiembre 2006 5.1

[38] A Runtime Assertion Checker for the Java Modeling Language, Yoonsik Cheon, Abril 2003 5

[39] Aplicación de JML a las prácticas de programación con Java, Antonio David Gomez Morillo, Antonio Menchen Penuela 6.2