SEP TNM

INSTITUTO TECNOLÓGICO DE CULIACÁN

UMAYA: Una herramienta para transformar programas en Ada a modelos Uppaal

TESIS

PRESENTADA ANTE EL DEPARTAMENTO ACADÉMICO DE ESTUDIOS DE POSGRADO DEL INSTITUTO TECNOLÓGICO DE CULIACÁN EN CUMPLIMIENTO PARCIAL DE LOS REQUISITOS PARA OBTENER EL GRADO DE

MAESTRO EN CIENCIAS DE LA COMPUTACIÓN

POR:

FRANCISCO HERNÁNDEZ GONZÁLEZ

INGENIERO EN SISTEMAS COMPUTACIONALES

DIRECTOR DE TESIS:

DR. RICARDO RAFAEL QUINTERO MEZA

CO-DIRECTOR DE TESIS:

DR. SERGIO CHRISTIAN HERRERA SALAZAR

CULIACÁN, SINALOA,2019

"2019, Año del Caudillo del Sur, Emiliano Zapata"

Culiacán, Sin., 7 de Agosto del 2019

DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN
OFICIO: DEPI:337/VIII/2019

ASUNTO: **Autorización Impresión**

ING. FRANCISCO HERNÁNDEZ GONZÁLEZ
ESTUDIANTE DE LA MAESTRÍA EN CIENCIAS DE LA COMPUTACIÓN
PRESENTE.

Por medio de la presente y en virtud de que ha completado los requisitos para el examen de grado de la **Maestría en Ciencias de la Computación**, se concede autorización para la impresión de la tesis titulada: **"UMAYA: UNA HERRAMIENTA PARA TRANSFORMAR PROGRAMAS EN ADA A MODELOS UPPAAL"** bajo la dirección del(a) **Dr. Ricardo Rafael Quintero Meza**

Sin otro particular reciba un cordial saludo.

ATENTAMENTE
Excelencia en Educación Tecnológica®

SEP TecNM
Instituto Tecnológico de Culiacán
División de Estudios de Posgrado e Investigación

M.C. MARÍA ARACELY MARTÍNEZ AMAYA
JEFE(A) DE LA DIVISIÓN DE ESTUDIOS DE POSGRADO E INVESTIGACIÓN

C.c.p. archivo

MAMA/lucy *

"UMAYA: UNA HERRAMIENTA PARA TRANSFORMAR PROGRAMAS EN ADA A MODELOS UPPAAL"

Tesis presentada por:

ING. FRANCISCO HERNÁNDEZ GONZÁLEZ

Aprobada en contenido y estilo por:

Dr. Ricardo Rafael Quintero Meza
Director de Tesis

Dra. María Lucía Barrón Estrada
Secretario

Dr. Ramón Zatarain Cabada
Vocal -1

M.C. Gloria Ekaterine Peralta Peñuñuri
Vocal -2

M.C. María Aracely Martínez Amaya
Jefe(a) de la División de Estudios de Posgrado e Investigación

Dedicatoria

Dedico esta tesis primero que nada a Dios, Conciencia Universal, Alá o como llamen a ese ser supremo que en ocasiones hemos sentido su presencia. A mis padres que sin ellos no hubiera podido ser quien soy ahora, gracias por todo el amor y cariño que siempre me han dado, por saber guiarme cuando lo he necesitado. A mi hermana por ser la molestia necesaria en mi vida. A mi novia por todo el apoyo que siempre me has dado, por ser el motor y la inspiración que me motiva a ser mejor cada día.

Agradecimientos

Agradezco en primera instancia al Instituto Tecnológico de Culiacán, por haberme abierto las puertas para poder realizar mis estudios de postgrado. A la Maestría en Ciencias de la Computación, por aceptarme y permitirme seguir creciendo como profesionista. A nuestra coordinadora Katty por siempre saber resolver las dudas que tenía en los diversos procesos del posgrado.

Agradezco enormemente a todos mis profesores quienes me enseñaron y me ayudador a crecer no solo como alumno si no también como persona, al Dr. Ramón Zatarain, Dra. Lucia Barrón, Dr. Héctor, sobre todo a mi Asesor y amigo Dr. Ricardo Quintero quien en conjunto con el Dr. Christian Herrera Salazar me guiaron y ayudaron en la conclusión de esta tesis.

A mis compañeros Polo, Emmanuel, Héctor, Ramón y Brandon con quienes compartí muy buenos momentos mientras cursaba mis estudios de posgrado, gracias por toda su ayuda, sus consejos y sobre todo su amistad.

Por último, agradezco a CONACYT por el apoyo económico con el cual me fue posible realizar mis estudios de posgrado.

Declaración de autenticidad

Por la presente declaro que, salvo cuando se haga referencia específica al trabajo de otras personas, el contenido de esta tesis es original y no se ha presentado total o parcialmente para su consideración para cualquier otro título o grado en esta o cualquier otra Universidad. Esta tesis es resultado de mi propio trabajo y no incluye nada que sea resultado de algún trabajo realizado en colaboración, salvo que se indique específicamente en el texto.

Francisco Hernández González. Culiacán, Sinaloa, México, 2019

Resumen

Garantizar la exactitud de sistemas críticos como los utilizados en la medicina, milicia, aeronáutica, etc. es de suma importancia. Es por ello que se utilizan técnicas de verificación de sistemas como la denominada verificación de modelos. Ada es el lenguaje de programación que se utiliza en su mayoría para el desarrollo de sistemas críticos, para aplicar la verificación de modelos a programas codificados en este lenguaje es necesario transformarlos a diagramas de modelos.

Actualmente existen muchas herramientas en la comunidad científica que ayudan a realizar el análisis estático de los programas desarrollados tanto en lenguaje Ada como algunos otros, la mayoría de este tipo de herramientas solo ayudan a detectar errores de tipo sintáctico y para garantizar la exactitud de los programas el apoyo es limitado, ya que Ada es un lenguaje utilizado por cierto número de desarrolladores que se desenvuelve en un nicho especializado, y no cuenta con suficientes herramientas de apoyo como con las que poseen los desarrolladores de otros lenguajes como Java y C++.

En este trabajo de tesis se propone la opción de transformar automáticamente dichos programas a modelos Uppaal, ya que actualmente en la mayoría de los casos dicha transformación se realiza de manera manual por los desarrolladores, lo que implica un doble esfuerzo. Se optó por la herramienta Uppaal, porque esta posee de manera nativa la posibilidad de implementar la verificación automática de modelos. Cuenta con una sección especializada donde fácilmente se pueden definir las diversas propiedades del sistema a verificar, lo que hace sumamente sencillo implementarla.

La herramienta UMAYA permite convertir un programa Ada automáticamente a un modelo Uppaal, logrando transformar la mayoría de las construcciones básicas de un programa Ada. Si bien el objetivo de la tesis se alcanzó, cabe mencionar que aún existe potencial de crecimiento en la herramienta propuesta, así como áreas de oportunidad mediante las cuales se puede mejorar y alcanzar objetivos a trabajo futuro.

Palabras clave

Ada

UMAYA

Uppaal

Verificación de modelos

Índice general

Índice de Figuras

Índice de tablas

Capítulo 1

1. Introducción

La verificación de modelos (Merz, 2001) es uno de los métodos más utilizados para asegurar la exactitud de los sistemas críticos y de tiempo real. Este tipo de sistemas son muy usados en sectores como la aviación, la milicia, la medicina y el espacio, donde la prevención de errores o posibles fallas son de suma importancia ya que pueden causar grandes pérdidas de dinero, dañar seriamente el ambiente o incluso ocasionar pérdidas de vidas humanas (Gang Tan, 2019).

Para el desarrollo de este tipo de sistemas se utiliza frecuentemente el lenguaje de programación Ada (Barnes J. , 2014) por sus características: lenguaje fuertemente tipado, concurrencia explícita, soporte para el diseño por contrato, paso de mensajes síncronos, objetos protegidos y programación no determinista lo que permite a los desarrolladores construir sistemas críticos y de tiempo real más robustos y confiables.

Existen varias herramientas que ayudan a verificar la exactitud de un programa en Ada, pero la mayoría de estas herramientas solo realizan análisis estático. El análisis estático es una técnica que analiza los programas sin necesidad de ejecutarlos. Existen dos actividades principales al momento de realizar un análisis estático, la primera es analizar la exactitud del programa en cuanto a la estructura gramatical del lenguaje de programación con un poco de alcance semántico limitado. Dos ejemplos típicos de estas herramientas son AdaControl (Adalog, 2019) y GNATCheck (AdaCore, 2019). La segunda, detectar errores en tiempo de ejecución como por ejemplo arreglos fuera de límites, desbordamientos aritméticos, divisiones por cero, etc., dos ejemplos prominentes de estas herramientas son Spark (Barnes J. G., 2003) y PolySpace (1994-2019 The MathWorks, 2019).

La mayoría de las herramientas mencionadas no verifican las propiedades temporales de un sistema. Verificar este tipo de propiedades es muy importante ya que mediante ello es posible revisar los diferentes estados de ejecución de un programa, usando la verificación de modelos se puede computarizar estos estados sin necesidad de ejecutar el programa. Es aquí donde

entra la importancia de usar los modelos Uppaal (G. Behrmann, 2004). Uppaal es un verificador de modelos ampliamente utilizado por la comunidad de desarrolladores de sistemas críticos y de tiempo real que utilizan una red de autómatas (Dill, 1994) para modelado y verificación de los mismos, lo que permite poder asegurar también las propiedades temporales de dichos sistemas. A su vez posee de manera nativa la posibilidad de implementar la verificación de modelos sin mayor problema. Cuenta con una sección especializada donde fácilmente se pueden definir las diversas propiedades del sistema a verificar, lo que hace sumamente sencillo implementarla.

Actualmente es difícil encontrar herramientas que puedan automáticamente transformar un programa Ada en un modelo Uppaal. Con el afán de poder verificar sus propiedades temporales, se desarrolló UMAYA, una herramienta capaz de transformar automáticamente un programa Ada a un modelo Uppaal, ofreciendo un camino para convertir las principales "construcciones" usadas en Ada (ciclos for, while, procedimientos, funciones, cláusulas with, if, etc) a sus equivalentes en Uppaal (localizaciones, bordes, guardias, etc) (G. Behrmann, 2004) respetando la sintaxis y semántica del programa original Ada.

1.1. Descripción del Problema

Sin duda uno de los requisitos no funcionales más importantes en los sistemas críticos, utilizados en la aviación, el espacio, la elaboración de sistemas ferroviarios, la milicia, etc. , es la seguridad, es por eso que es necesario el apoyo de herramientas que ayuden a garantizar dicha seguridad al momento de programar un sistema software. En la mayoría de las ocasiones Ada es el lenguaje de programación frecuentemente usado en estos ámbitos, y el principal problema que se puede encontrar al momento de convertir programas Ada a modelos Uppaal es el respetar la semántica de dichos programas al transformarlos a un diagrama de modelo.

Utilizando la técnica de verificación de modelos, es posible poder comprobar la exactitud de los programas Ada, para ello una opción sería transformar dichos programas Ada a modelos

Uppaal. Habitualmente la transformación de programas Ada a modelos Uppaal se realiza manualmente por los desarrolladores que codifican el programa, lo que conlleva un doble esfuerzo y mayor tiempo al realizar la verificación de los mismos.

Actualmente es difícil encontrar herramientas que ayuden a realizar dicha transformación de manera automática, lo cual ayudaría a los programadores a evitar el crear ellos el modelo Uppaal manualmente. Por lo que contar con una herramienta que realice ese trabajo de manera automatizada, respetando la sintaxis y semántica de los programas originales sería de gran utilidad, reduciendo así los tiempos de verificación.

1.2. Objetivo General

Desarrollar una herramienta en el lenguaje de programación Java (Oracle, 2019) capaz de transformar automáticamente un programa Ada a un modelo Uppaal, respetando la sintaxis y semántica del programa original con el objetivo de facilitar el trabajo a los desarrolladores de sistemas críticos y tiempo real al momento de implementar técnicas como la verificación de modelos.

1.3. Objetivos Específicos

- Procesar y analizar el archivo fuente de código Ada.
- Transformar el archivo Ada a archivo especial XML que respete la especificación de la interfaz semántica de Ada.
- Procesar la información obtenida del archivo XML para crear las diferentes clases y objetos de programación necesarios para realizar la transformación del programa a modelo Uppaal.
- Transformar la información procesada en un XML el cual pueda ser interpretado por Uppaal para generar el modelo equivalente al programa Ada original.

1.4. Hipótesis

Utilizando la herramienta UMAYA se garantiza la transformación correcta de un programa Ada a un diagrama de modelo Uppaal, respetando principalmente su semántica. Para así ayudar a la implementación de técnicas como la verificación de modelos.

1.5. Organización de la Tesis

El proceso de investigación realizado para elaborar la herramienta llamada UMAYA, se encuentra organizado en los siguientes capítulos:

El capítulo 2 muestra las bases teóricas necesarias para poder llevar a cabo la construcción de la herramienta, donde se tratarán temas como la programación en el lenguaje Ada, XML, la especificación de la interfaz semántica de Ada, el modelado Uppaal y la verificación de modelos.

En el capítulo 3 se describe el estado del arte de algunas herramientas que han intentado alcanzar los mismos objetivos o similares de nuestro proyecto, que es lograr la transformación automática de un programa Ada a modelo Uppaal.

El capítulo 4 detalla el proceso mediante el cual se desarrolló la herramienta, describe las diversas fases y componentes creados para poder llevar a cabo nuestro proyecto.

El capítulo 5 muestra el resultado de la prueba realizada, cuyo propósito es validar la hipótesis previamente estipulada.

Por último, en el capítulo 6 presenta las conclusiones, el aporte actual y de futuros trabajos en caso de continuarse con el tema de investigación de esta tesis.

Capítulo 2

2. Marco teórico

En este capítulo se abordarán las bases teóricas necesarias para poder llevar a cabo la investigación de este proyecto de tesis.

2.1. Programación en el lenguaje Ada

La historia de Ada como un lenguaje de programación se remonta al año de 1974 cuando el Departamento de la defensa de los Estados Unidos, a razón de que llevaban mucho tiempo gastando en el desarrollo de software principalmente en el área de sistemas integrados, decidió patrocinar un proyecto en el que después de llevarse a cabo una serie de fases de definición de requisitos muy competitiva, desarrollo en paralelo y continua evaluación, terminó convirtiéndose en el estándar original de Ada en 1983. El equipo de desarrollo de Ada estaba conformado por el CII Honeywell Bull (BULL computers chronological history, 2019) en Francia bajo el liderato de Jean D Ichbiah.

El nombre del lenguaje de programación fue tomado de Augusta Ada Byron, condesa de Lovelace (1815-52), hija del poeta Lord Byron y asistente del matemático inglés Charles Babbage. Fue gracias a las aportaciones que le hizo al trabajo de dicho matemático, en la llamada máquina analítica, que se le consideró la primera programadora de la historia, sirviendo así como inspiración para elegir el nombre para el lenguaje de programación recién creado.

A lo largo del tiempo han existido diversas versiones del lenguaje de programación Ada, cada una aportando algo nuevo y enriqueciendo aún más el lenguaje. Por características como las que se mencionan más adelante, es el lenguaje que comúnmente se utiliza para el desarrollo de sistemas críticos y de tiempo real.

2.1.1. Versiones del lenguaje Ada

- Ada 83 mostró a los programadores como podría organizarse una programación enfocada a desarrollos sumamente grandes (paquetes, lenguaje fuertemente tipado, privacidad). Demostró que errores tan básicos como los índices fuera de rango, podían ser controlados fácilmente con una declaración apropiada de las variables y verificación de las restricciones, además introdujo la programación concurrente al lenguaje de programación convencional.
- Ada 95 se vio beneficiada con la intrusión de diversas técnicas de programación orientada a objetos, ideas exitosas sobre polimorfismo y una dinámica interacción entre un lenguaje fuertemente tipado concurrente.
- Ada 2005 agregó la unificación de un lenguaje fuertemente tipado y concurrente a un nuevo modelo de herencia múltiple. Adicionó también un extenso contenedor de bibliotecas que agregaban una basta funcionalidad extra al lenguaje.
- Ada 2012 introduce nuevos caminos en la forma de realizar la programación como los son los contratos, los cuales permiten especificar precondiciones o postcondiciones a nuestro programa, cuenta también con tareas que ayudan a reconocer arquitecturas multinúcleo, se agregaron y mejoraron bibliotecas al contenedor de bibliotecas de Ada.

2.1.2. Acerca de Ada

En esta sección se presenta información que es importante conocer acerca del lenguaje de programación Ada para así tener un mejor contexto del mismo.

Objetivos Claves

Ada es un lenguaje sumamente amplio, que, si bien puede ser utilizado para fines didácticos o para desarrollar programas cortos como Pascal, es un lenguaje al nivel de C++ en cuestión de complejidad, pero lo que más lo diferencia de éste, es la importancia que Ada le da a la integridad y la legibilidad de sus programas. Algunos de los aspectos a los que Ada toma mayor importancia son:

- Legibilidad – cuando se desarrolla profesionalmente, son más las ocasiones en que se lee un programa que lo que se escribe. Es por ello que es importante evitar el estilo de codificación de manera encriptada, como por ejemplo nombrar variables de la forma VAR1. Si bien esto ayuda a codificar de manera más rápida, resulta casi imposible leer y entender dicho programa a otra persona que no haya sido el autor original del código.
- Fuertemente tipado – Esto quiere decir que en Ada cada elemento está definido con un conjunto de valores bien establecido, lo que ayuda a evitar confusión entre conceptos lógicamente distintos. Como consecuencia muchos de los errores son detectados por el compilador.
- Programación a lo grande – Ada cuenta con mecanismos de encapsulamiento, compilación en paralelo y una administración de bibliotecas que hacen posible realizar programas sumamente portables y mantenibles de cualquier tamaño.
- Manejo de Excepciones – Siempre es necesario tener una manera fácil y segura de poder manejar las posibles fallas que se presenten en un programa. Ada cuenta con una capa muy bien definida para el manejo de excepciones.
- Abstracción de Datos – Una portabilidad y mantenibilidad extra es posible alcanzar si logramos mantener los detalles de la representación de los datos, separada de las especificaciones lógicas de las operaciones de los datos. Por lo que Ada fomenta este tipo de prácticas abstractas.
- Programación Orientada a Objetos – con el fin de fomentar el reúso de código ya probado, Ada permite toda la flexibilidad asociada con la programación orientada a objetos (Meyer, 1997), como lo es la herencia, polimorfismo, abstracción, etc.
- División en tareas – En la mayoría de las aplicaciones es importante poder dividir su funcionamiento en diversas tareas que puedan interactuar entre sí. Es por ello que Ada fomenta diferentes vías para poder crear múltiples tareas que convivan entre ellas.
- Unidades Genéricas – En muchas ocasiones la parte lógica de los programas es independiente de la manera en que se manipulan los datos. Por lo que es necesario

crear mecanismos que puedan servir como plantillas para la elaboración de objetos más complejos. Esto es sumamente útil, por ejemplo, para la creación de bibliotecas.

- Comunicación – Los programas no viven aislados, por lo que es de suma importancia contar con diversas vías de comunicación para poder interactuar con otros programas, sistemas, servicios, etc. Sin importar si estos fueron incluso escritos en otro lenguaje.

Estructura General

En la actualidad uno de los objetivos principales de los Ingenieros de Software es el reúso de código ya existente que funcione correctamente, para evitar la generación de código nuevo innecesario. Ada reconoce esta situación e introduce el concepto de bibliotecas.

Un programa de Ada es reconocido como un subprograma principal o una biblioteca en sí, que realiza peticiones a los servicios de otras bibliotecas. El subprograma principal toma la forma de un procedimiento con su respectivo nombre. Los servicios de las bibliotecas pueden ser otros procedimientos o funciones, pero generalmente son tratados como paquetes. Un paquete es un grupo de elementos relacionados que pueden ser otros subprogramas pero que también puede contener otras entidades. En la Figura 2-1, se muestra un ejemplo de un programa Ada.

```
with Sqrt, Simple_IO;
procedure Print_Root is
   use Simple_IO;                          -- declarations here
   X: Float;
begin
   Get(X);                                 -- statements here
   Put(Sqrt(X));
end Print_Root;
```

Figura 2-1 Ejemplo 1 de un programa Ada. *(Barnes J. , 2014)*

Al analizar dicha Figura se observa como el subprograma principal es representado como un procedimiento simple. Mediante las cláusulas with se agrega visibilidad al programa para

poder utilizar los servicios de funciones (Sqrt), procedimientos o paquetes (Simple_IO). Al revisar la zona de las declaraciones se observan las variables que utilizará el programa, así como también se indica los paquetes que se utilizarán en el mismo. Dentro del cuerpo del programa, se encuentran las diversas sentencias que proporcionan la funcionalidad que se requiere.

En la Figura 2-2 se muestra un ejemplo sencillo pero un poco más completo de lo que es un programa Ada.

```
with Sqrt, Simple_IO;
procedure Print_Roots is
   use Simple_IO;
   X: Float;
begin
   Put("Roots of various numbers");
   New_Line(2);
   loop
      Get(X);
      exit when X = 0.0;
      Put(" Root of "); Put(X); Put(" is ");
      if X < 0.0 then
         Put("not calculable");
      else
         Put(Sqrt(X));
      end if;
      New_Line;
   end loop;
   New_Line;
   Put("Program finished");
   New_Line;
end Print_Roots;
```

Figura 2-2 Ejemplo 2 de un programa Ada. *(Barnes J. , 2014)*

En ella se aprecia una estructura más compleja del programa, con diversas llamadas a funciones e incluso la ejecución de un ciclo. Conocer cómo se conforma un programa Ada será de utilidad más adelante.

2.2. XML

XML (W3C , 2019) nació a partir de un lenguaje desarrollado por IBM llamado lenguaje de marcado generalizado (GML por sus siglas en inglés). GML surgió por la necesidad que se tenía de manejar grandes cantidades de información las cuales tenían que compartirse con diversos sistemas operativos o plataformas. En 1986 este lenguaje fue estandarizado por ISO (ISO, 2019) dando así lugar al lenguaje de marcado generalizado estándar (SGML por sus siglas en inglés) del cual el lenguaje de marcado extensible (XML por sus siglas en inglés) es un subconjunto.

Los documentos XML están compuestos por unidades de almacenamiento denominadas entidades, las cuales contienen los datos analizados o a analizar. Los datos analizados están formados por caracteres del lenguaje XML, caracteres de los datos en sí y etiquetas.

Las etiquetas representan la estructura lógica del documento, la forma en que se almacenan los datos en el mismo. XML proporciona un mecanismo para imponer restricciones en el diseño de almacenamiento y la estructura lógica.

2.2.1. Origen y Objetivos de XML

XML fue desarrollado por un grupo de trabajo patrocinado por el Consorcio Mundial de la Red (W3C por sus siglas en inglés) en 1996. Los objetivos para los que fue diseñado XML son:

- XML podrá usarse directamente en internet.
- XML soportará gran variedad de aplicaciones.
- XML deberá ser compatible con el SGML.
- Deberá ser sencillo codificar programas que procesen documentos XML.
- El número de características opcionales de XML se deben de mantener al mínimo, de preferencia 0.
- Los documentos XML deben de ser claros y legibles.
- Un documento XML debe generarse rápidamente.
- El diseño del documento XML debe ser conciso y formal.

- Los documentos XML deben de ser fáciles de generar.
- La brevedad en un archivo XML es de mínima importancia.

2.2.2. Documentos XML válidos y bien formados.

Cuando un documento XML no contiene errores de sintaxis se dice que está bien formado (*Well-Formed*). Para lograr esto es necesario tomar en cuenta los siguientes puntos:

- Los nombres de los elementos y sus atributos, en caso de tenerlos, deben estar escritos correctamente.
- Los valores de los atributos deben estar escritos entre comillas.
- Cada atributo de un elemento se debe separar por un espacio en blanco.
- Solo puede existir un elemento raíz.
- Exceptuando la raíz, todo elemento debe tener un elemento padre.
- Los elementos deben tener siempre una etiqueta de apertura y otra de cierre.
- El anidado entre etiquetas debe ser correcto.
- Las instrucciones a ser procesadas deben estar escritas correctamente.
- La primera línea en el documento debe ser la declaración de que es un documento XML.
- Los comentarios y las secciones CDATA deben estar escritas correctamente.

Así mismo, se dice que un documento XML es válido cuando no contiene ningún error de sintaxis y además no incumple ninguna de las normas definidas en su estructura. Dicha estructura está definida por los métodos de definición del tipo de documento (DTD por sus siglas en inglés), el esquema XML y el lenguaje regular para XML siguiente generación (RELAX NG por sus siglas en inglés).

En la Figura 2-3, se observa un ejemplo de un archivo XML bien formado.

```xml
<?xml version="1.0" encoding="UTF-8" ?>
<alumnos>
    <alumno id="1">
        <nombre>Mario</nombre>
        <apellido>Perez</apellido>
    </alumno>
    <alumno id="2">
        <nombre>Abril</nombre>
        <apellido>Cazares</apellido>
    </alumno>
    <alumno id="3">
        <nombre>Juan</nombre>
        <apellido>Bastida</apellido>
    </alumno>
</alumnos>
```

Figura 2-3 Ejemplo de un documento XML bien formado.

2.3. Especificación de la interfaz semántica de Ada (ASIS)

La Especificación de la interfaz semántica de Ada (ASIS por sus siglas en inglés), es una interfaz mediante la cual es posible proporcionar a cualquier aplicación o herramienta información sobre un entorno Ada. Un entorno Ada cuenta con cierta información semántica y sintáctica, por medio de la interfaz ASIS (Roby, 2019), esta información puede ser accedida por diversas herramientas CASE.

La interfaz ASIS está conformada por diversos paquetes que ayudan a obtener la información necesaria de un entorno Ada particular, siendo el paquete ASIS la raíz principal. Este paquete está conformado por un conjunto de tipos, subtipos y subprogramas que permiten realizar consultas al entorno de compilación Ada del cual se necesita información.

Algunas abstracciones importantes que se manejan en la interfaz ASIS son: Contexto, Elemento y Unidad de compilación. El tipo *Contexto* ayuda a la interfaz a identificar la unidad de compilación que será analizada en el entorno de compilación Ada. El tipo *Elemento* es una abstracción de entidades dentro de un árbol de sintaxis lógica Ada. El tipo *Unidad* de compilación es una abstracción que representa la unidad de compilación Ada que se analiza. Además, existen dos conjuntos de tipos de enumeración llamados tipos de elementos y tipos de unidad. Los tipos de elementos son un conjunto de tipos de enumeración que proveen un

mapeo de la sintaxis de Ada. Los tipos de unidad son un conjunto de tipos de enumeración que describen diferentes tipos de unidades de compilación.

A continuación, en la Figura 2-4, se muestra la interacción de un entorno Ada con la interfaz ASIS.

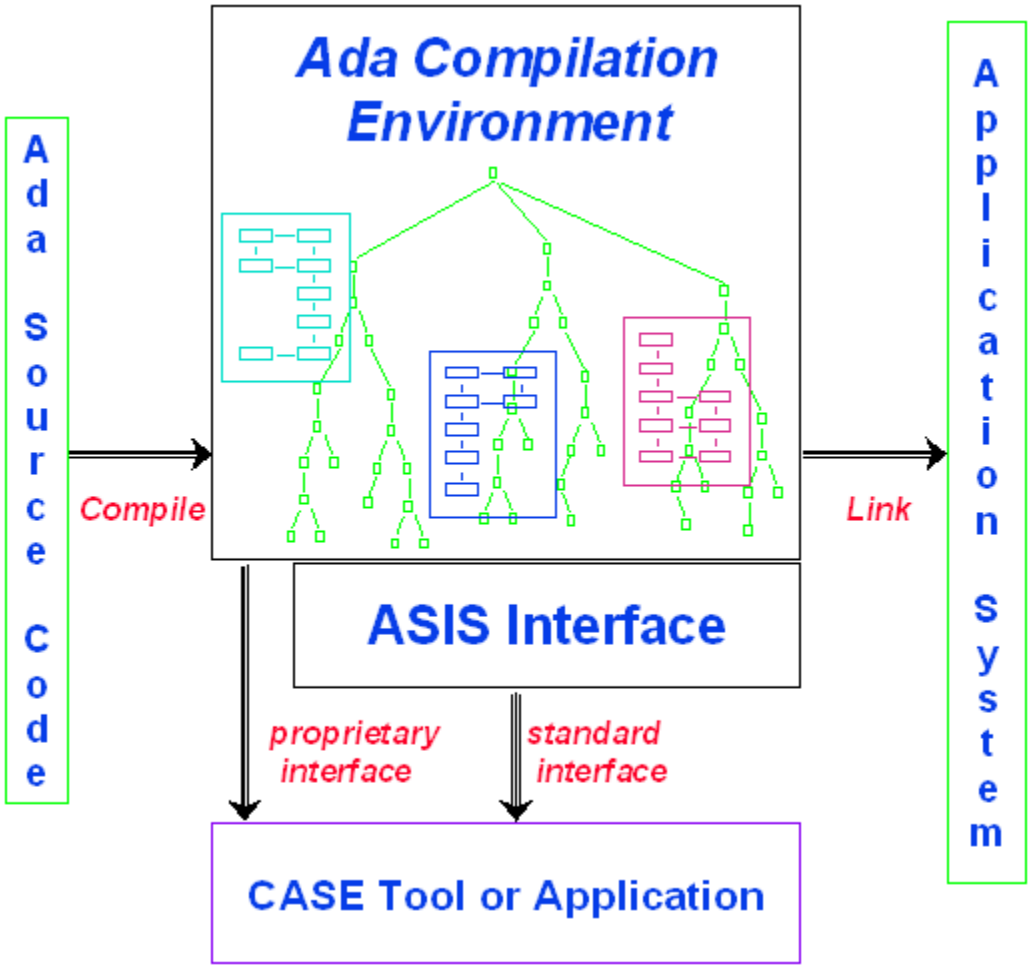

Figura 2-4 ASIS como interfaz para un entorno de compilación Ada. (Roby, 2019)

La Figura 2-5 representa la arquitectura del paquete raíz de la interfaz ASIS.

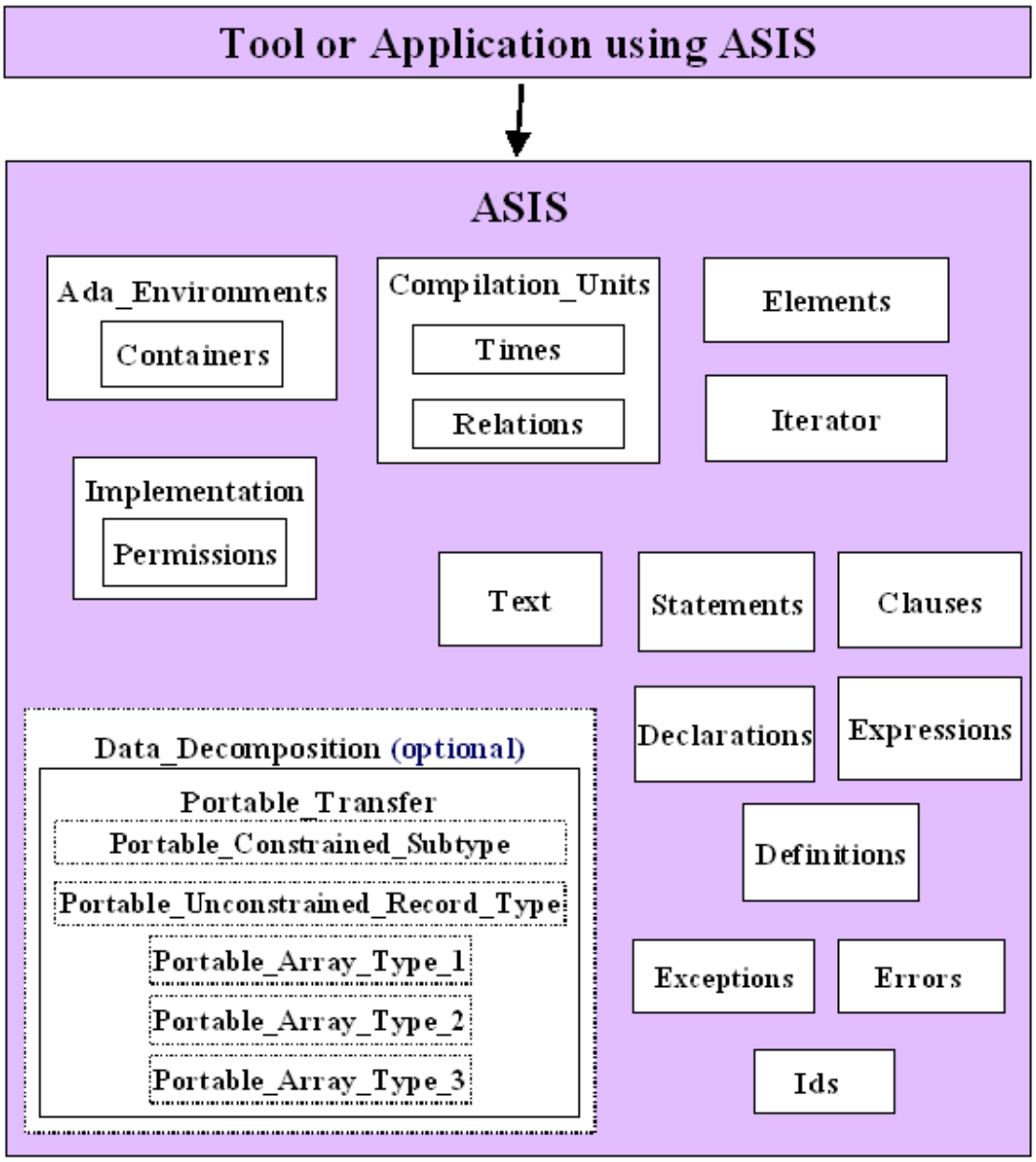

Figura 2-5 Arquitectura del paquete ASIS. (Roby, 2019)

2.4. Uppaal

Uppaal (G. Behrmann, 2004) es una caja de herramientas utilizada para la validación, por medio de simulación gráfica; y verificación de manera automática de modelos de sistemas en tiempo real. Fue desarrollado por la Universidad de Uppsala y la Universidad de Aalborg. La herramienta está diseñada para poder verificar sistemas que puedan ser modelados como una red de autómatas (Dill, 1994) que utilicen elementos como variables enteras, estructuras de tipos de datos, funciones definidas por el usuario y canales de sincronización.

La primera versión de Uppaal se lanzó en 1995, pero es la versión 4.0 la que se utilizará para la elaboración de la herramienta UMAYA. La versión 4.0 de Uppaal requiere que Java Runtime Environment (Oracle, 2019) 5 o superior este instalado en la computadora para su ejecución.

Al utilizar Uppaal para la verificación de sistemas, la idea es modelarlos usando autómatas para así poder realizar una simulación y verificar las diferentes propiedades de los mismos. Un autómata tiene un conjunto de estados finitos en determinado tiempo. En Uppaal los *clocks* representan el tiempo en el proceso, las *locations* representan algún proceso en específico y los *edges* las transiciones entre ellos. Un sistema entonces es una red de *locations* cuya interacción entre ellas define el comportamiento del sistema. La simulación paso a paso consiste en ejecutar el sistema interactivamente para verificar que trabaje adecuadamente. Después mediante el verificador se revisan las propiedades de accesibilidad, por ejemplo, que se hayan alcanzado todos los posibles estados a los que el sistema puede llegar. A esto se le conoce como verificación de modelos, la cual es la exhaustiva búsqueda por cubrir todos los posibles estados dinámicos en el comportamiento de un sistema. En la Figura 2-6 se muestra la ventana principal de Uppaal.

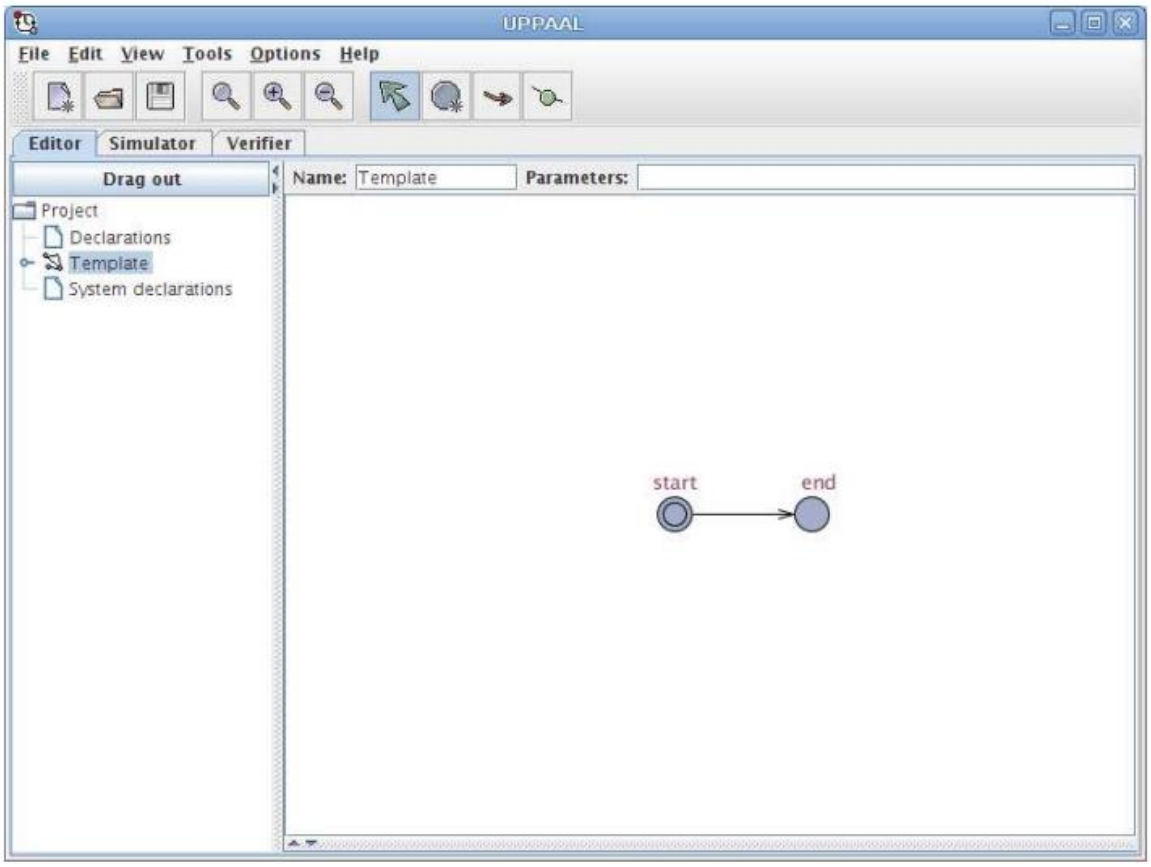

Figura 2-6 Ventana principal de Uppaal. (Uppsala Universitet, 2019)

En ella se observan las tres etiquetas con los componentes principales de Uppaal: el editor, el simulador y el verificador.

En la vista editor, el objetivo es realizar *templates* que representen procesos los cuales en conjunto representen a un sistema entero. Para ello se utilizan diferentes elementos como los ya mencionados anteriormente: *clocks*, *locations*, *edge* etc.

Se observa como en la imagen 2-6 se encuentran la *location start* que representa el inicio del autómata, seguida de un *edge* (transición) la cual conlleva a la siguiente *location end* que representa el final de la ejecución del autómata. Básicamente, todo sistema se puede representar como un conjunto de estados y transiciones los cuales pueden representarse en Uppaal con estos elementos para su simulación. En la Figura 2-7 se muestra la vista simulador.

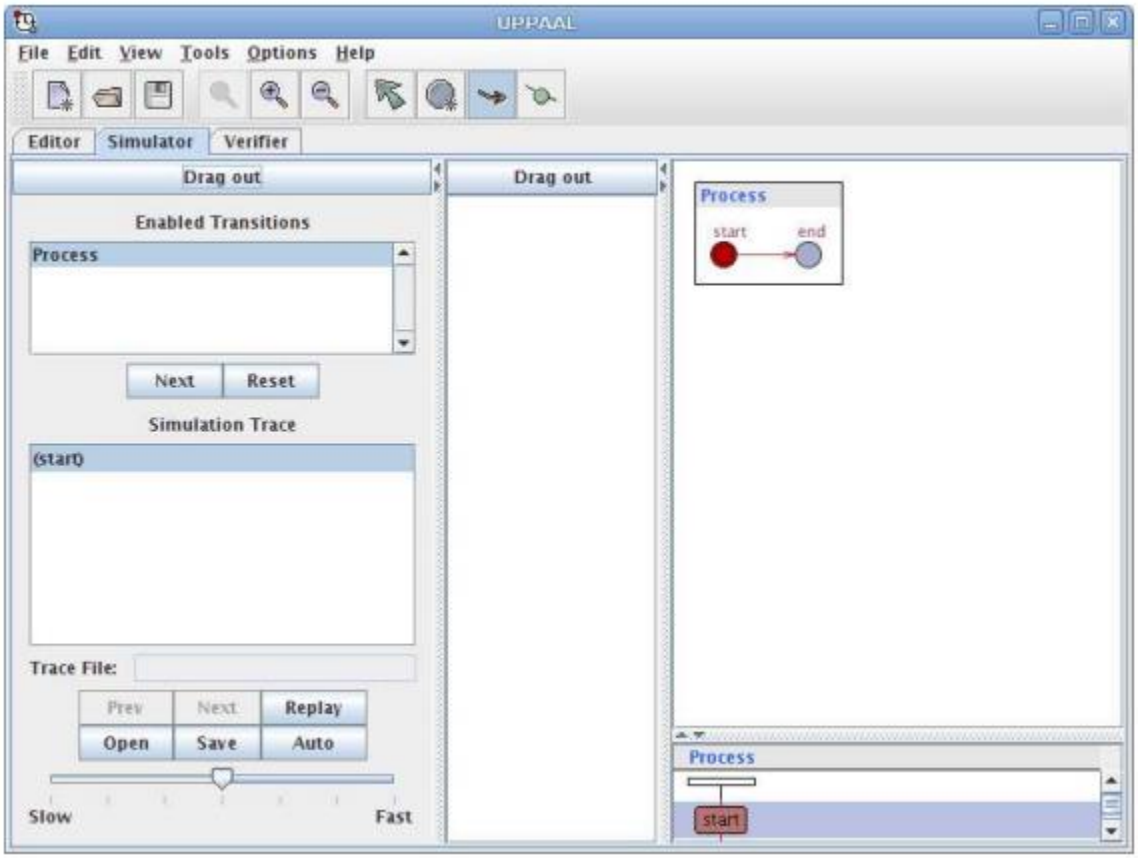

Figura 2-7 Vista simulador de Uppaal. (Uppsala Universitet, 2019)

En la parte izquierda superior de la vista simulador, se aprecian las distintas transiciones que contenga el autómata, así como en la parte izquierda inferior la trazabilidad de las mismas. En la parte derecha se observa el autómata ejecutándose, a su vez en la parte inferior derecha se aprecian los diferentes cambios de estado que el sistema va experimentando a lo largo de su ejecución. En la Figura 2-8, se observa la última de las vistas de Uppaal, la vista verificador.

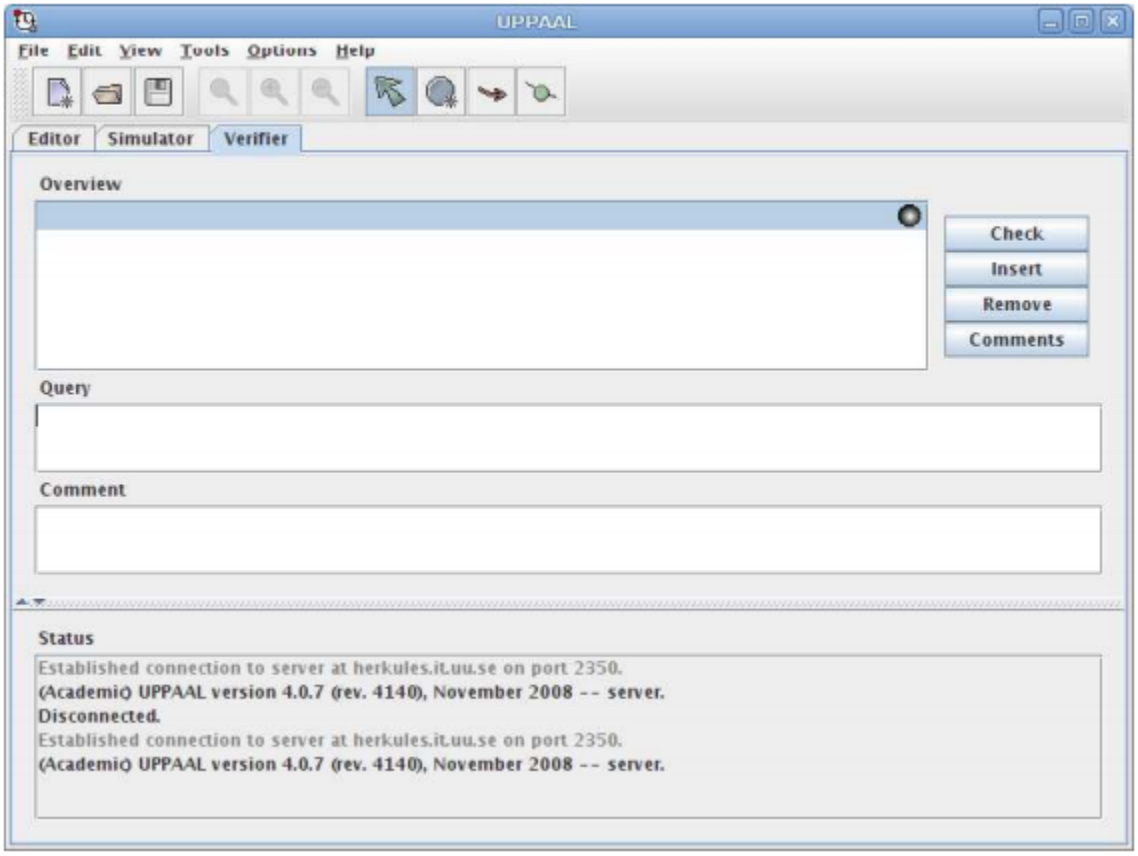

Figura 2-8 Vista verificador de Uppaal. (Uppsala Universitet, 2019)

En la vista verificador, se especifican consultas al sistema. Es aquí donde se definen algunas reglas o aserciones para verificar algunas propiedades con las que deba cumplir el sistema. Al dar clic en el botón check, Uppaal verificará si al ejecutarse el autómata, cada una de las reglas que hayan definido en esta pantalla se cumplen o no. Si el foco a lado de cada regla se muestra en verde, significa que el sistema cumplió con la propiedad asignada a esa regla, en caso contrario el foco se muestra en color rojo. Es gracias a esta propiedad de Uppaal que se pueden aplicar técnicas como la verificación de modelos a los sistemas.

En las Figuras 2-9 y 2-10, se observan dos procesos simples que interactúan entre sí y su representación equivalente en un modelo Uppaal.

Process 1

```
idle:
 req1=1;
want:
 turn=2;
wait:
 while(turn!=1 && req2!=0);
CS:
//critical section
 job1();
 req1=0;
//and return to idle
```

Process 2

```
idle:
 req2=1;
want:
 turn=1;
wait:
 while(turn!=2 && req1!=0);
CS:
//critical section
 job2();
 req2=0;
//and return to idle
```

Figura 2-9 Procesos que interactúan entre sí. (Uppsala Universitet, 2019)

Cuando el primer proceso se ejecuta pasa del estado *idle* al estado *want* ocasionando que la variable req1 cambie su valor, al salir del estado *want* la variable turn también cambia su valor lo que produce que se cumpla la condición estipulada por la sentencia while que lleva al programa hacia un nuevo estado, el estado *wait*.

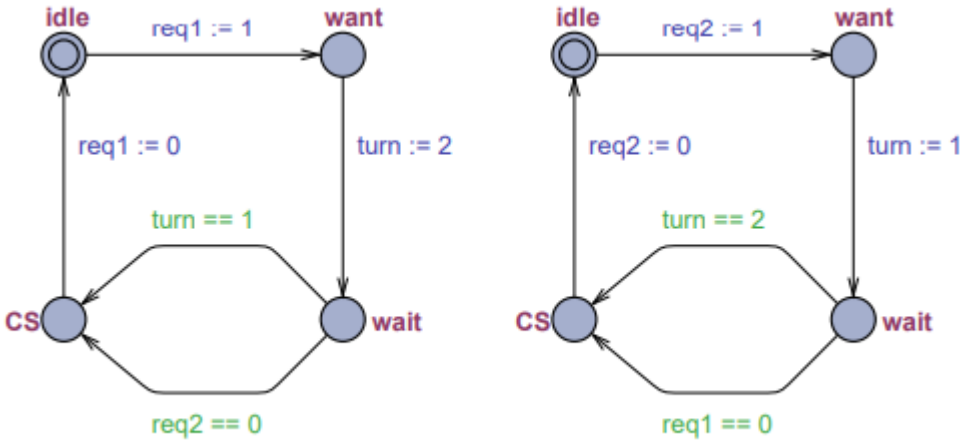

Figura 2-10 Modelo Uppaal equivalente. (Uppsala Universitet, 2019)

En ese momento se dispara también el siguiente proceso, siguiendo los mismos pasos del primer proceso hasta el momento en que alcanza el estado *wait*. Al llegar el segundo proceso al estado *wait* se modificó el valor de la variable turn, esto ocasionó que la condición en la sentencia while del primer proceso ya no se cumpla, por lo que paso del estado *wait* al estado *cs*. En ese cambio de estado las variables turn y req2 vuelven a cambiar su valor y se ejecutan la sentencia que se encuentran en la sección critica del primer proceso, al cambiar los valores de las variables turn y req2, ocasiono que el segundo proceso se desbloqueara y ejecutara también su sección critica. Es así como los dos procesos interactúan entre si bloqueando y desbloqueando sus estados mutuamente.

2.5. Verificación de Modelos

Antes de presentar la verificación de modelos, se debe hablar un poco sobre la verificación de sistemas. Hoy en día es una realidad que la confianza en las Tecnologías de la Información y Comunicaciones (TIC) ha crecido muy rápidamente.

Día a día se interactúa con diferentes sistemas, desde el sistema operativo del teléfono celular, el sistema con el que se cobra en el super mercado, así como sistemas de suma importancia como los utilizados en sectores como la medicina, del cual dependen innumerables vidas. Este tipo de sistemas son los denominados sistemas críticos, sistemas en los que cualquier fallo puede producir enormes pérdidas, ya sea monetariamente, daños al medio ambiente o vidas humanas.

Es aquí donde entra la importancia de la verificación de sistemas y la verificación de modelos. La verificación de sistemas, es el hecho de crear o diseñar un producto tomando en cuenta desde su concepción que cumpla con ciertas propiedades ya definidas con anterioridad. Las propiedades a verificar pueden ser muy básicas como, por ejemplo, el asegurarse que el sistema no entre en un estado de bloqueo. La mayoría de las propiedades se obtienen de las especificaciones del sistema, estas especificaciones prescriben que es lo que tiene y no tiene que hacer el sistema. Y asegurar esto es básicamente la función de cualquier actividad de verificación de un sistema. Aun si se encuentra un defecto que no estaba dentro de las especificaciones podemos decir que el sistema es “correcto”, siempre y

cuando cumpla con todas las especificaciones. Por lo que la exactitud de un sistema es siempre relativa a las especificaciones del mismo, no es una propiedad absoluta del sistema en sí.

En la Figura 2-11, se ilustra el proceso de verificación de un sistema.

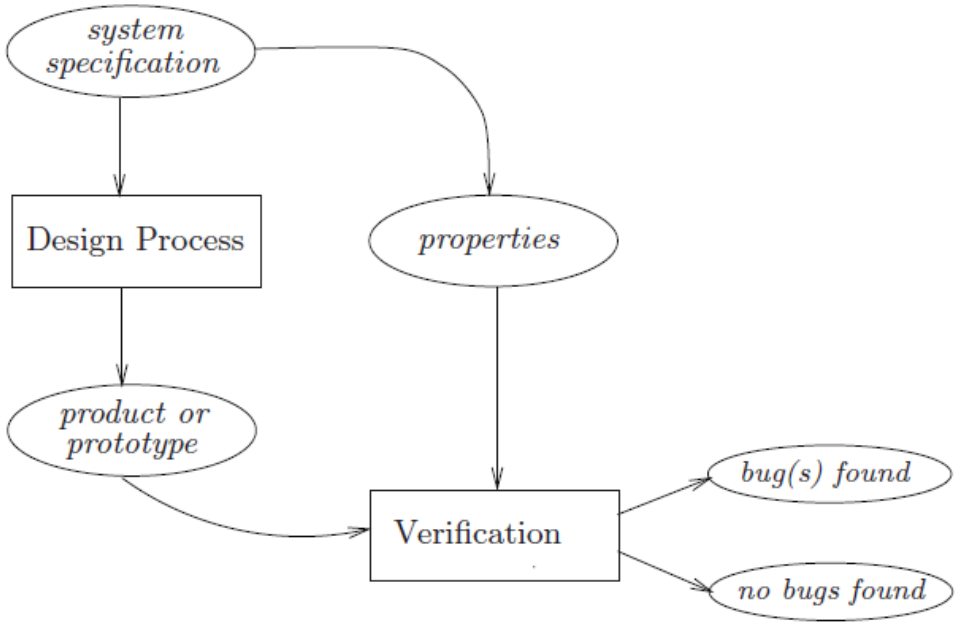

Figura 2-11 Proceso de Verificación. (Katoen, 2008)

Ya que se entiende lo que es la verificación de sistemas, se tocará un poco el tema de la verificación de modelos. La verificación de modelos es una técnica de verificación que empieza con una especificación formal del sistema. Para poder realizar la especificación formal de un sistema, es necesario aplicar métodos formales. Los cuales ofrecen un gran potencial para poder integrar de forma temprana la verificación al proceso de diseño, proveen también más efectividad a las técnicas de verificación y reducen el tiempo de la misma.

El uso de métodos formales es considerado como el aplicar las matemáticas para el modelado y diseño de los sistemas TIC. Lo cual apunta a establecer la exactitud de los sistemas con el rigor de las matemáticas. El gran potencial de utilizar métodos formales para la verificación de complejos sistemas de software ha llevado a muchos ingenieros a utilizarlos concurrentemente. Por lo que hoy en día los métodos formales son altamente recomendados

como técnica de verificación para el desarrollo de sistemas críticos seguros, de acuerdo con el estándar de las mejores prácticas de la Comisión Electrotécnica Internacional (IEC por sus siglas en inglés) y estándares de la Agencia Espacial Europea (ESA por sus siglas en inglés). El reporte de resultados de investigación de la Autoridad Federal de Aviación (FAA por sus siglas en inglés) y la Administración Nacional de Aeronáutica y Espacio (NASA por sus siglas en inglés).

Existen también técnicas de verificación basadas en modelos, estas técnicas se basan en modelos que describen el posible comportamiento de un sistema con una precisión matemática no ambigua. En esta categoría es donde se encuentra la verificación de modelos.

La verificación de modelos es una técnica de verificación que explora todos los posibles estados de un sistema a "fuerza bruta". Muy parecido a un sistema de ajedrez que explora todos los posibles movimientos de una jugada, un verificador de modelos es una herramienta de software que examina todos los posibles escenarios de una manera sistemática. De esta manera puede garantizar que el sistema cumple con cada una de las propiedades definidas en su especificación. En la Figura 2-12, se visualiza el enfoque de la verificación de modelos.

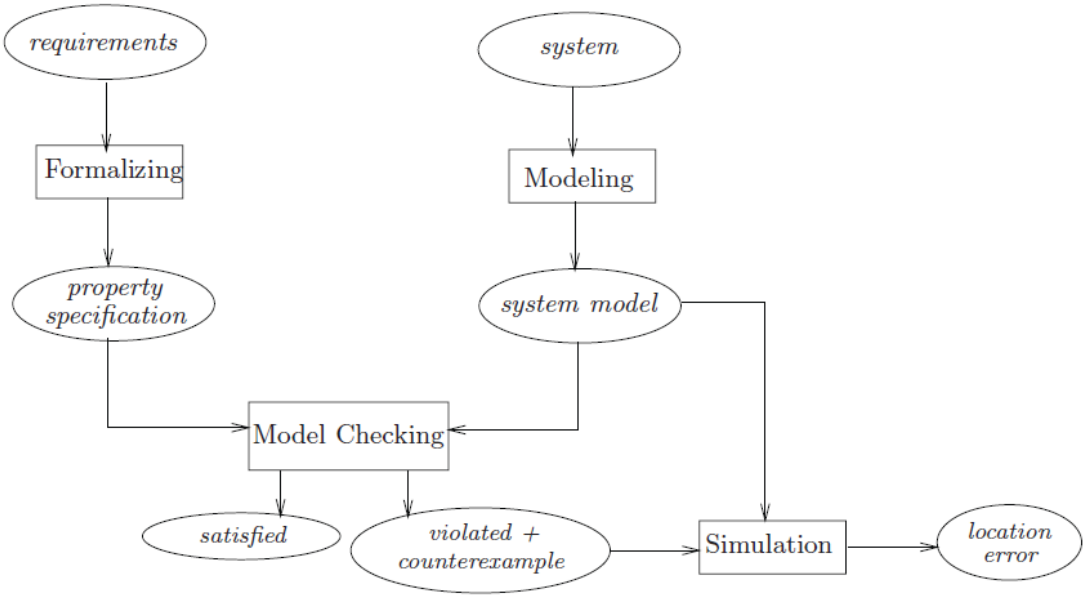

Figura 2-12 Verificación de modelos. (Katoen, 2008)

La verificación de modelos se establece en tres fases:

- Fase de Modelado:
  - Modelar el sistema utilizando el lenguaje correspondiente al verificador de modelos que se utilice.
  - Como primera evaluación y validación rápida del modelo, realizar algunas simulaciones.
  - Formalizar las propiedades que vamos a verificar, utilizando el lenguaje para la especificación de propiedades.
- Fase de Ejecución:
  - Ejecutar el verificador de modelos, para verificar la viabilidad de las propiedades en el modelo del sistema.
- Fase de Análisis:
  - ¿Propiedad satisfecha? Entonces, verificar la siguiente propiedad.
  - ¿Propiedad no alcanzada? Entonces:
    1. Analizar el contraejemplo generado por simulación.
    2. Refinar el modelo, diseño o propiedad.
    3. Repetir el proceso completo.
  - ¿Desbordamiento de memoria? Entonces, reducir el modelo e intentar de nuevo.

En general, el proceso de verificación de modelo se define en estas tres grandes fases. La aplicación de este tipo de técnicas, es lo que aporta un valor significativo al resultado del trabajo de esta tesis.

2.6. Cierre

En este capítulo se presenta información sobre algunos conceptos y lenguajes informáticos, así como también herramientas que ayudarán al entendimiento de los siguientes capítulos en los que se detalla la elaboración de la herramienta desarrollada en este trabajo de tesis.

Capítulo 3

3. Estado del arte

En esta sección se describen algunos trabajos de investigación que sirvieron de base para este trabajo de tesis. Se mencionan algunas herramientas que intentaron obtener modelos a partir de programas Ada, con la intención también de poder aplicar en ellos técnicas de verificación de modelos.

3.1. ATOS

ATOS (Faria J.M., 2012) es una herramienta que automáticamente es capaz de obtener modelos en SPIN (Holzmann, 1997) de programas Ada. ATOS es capaz de obtener propiedades incluso de notaciones inspiradas en las utilizadas en el lenguaje SPARK. El objetivo de la herramienta es ayudar a la verificación de programas Ada basándose en la verificación de modelos.

ATOS incluye sus propios algoritmos de verificación de modelos para garantizar la exactitud de los mismos.

En la Figura 3-1, podemos observar la estructura de simulación y verificación de SPIN, donde se aprecia como ATOS utiliza el verificador SPIN el cual genera sus modelos en un metalenguaje de procesos (PROMELA (Holzmann, 1997) por sus siglas en inglés), el lenguaje de modelado de SPIN. Las especificaciones de diseño que sea necesario hacer al momento de verificar la exactitud de los modelos, pueden ser hechas mediante la sintaxis estándar de una lógica temporal lineal (LTL por sus siglas en inglés) o mediante aserciones en PROMELA.

Una vez que se determinaron las especificaciones, se pasa a la siguiente etapa la cual es la detección y corrección de los errores de sintaxis. Cuando todo error de sintaxis ha sido corregido, se realiza una simulación interactiva del modelo en la que el generador del verificador procede a una ejecución y optimización al vuelo, con la cual si se encuentra algún

contraejemplo que no cumpla con las especificaciones definidas, este es analizado e inyectado nuevamente a la simulación para tratar de identificar detalladamente ese caso en particular y removerlo.

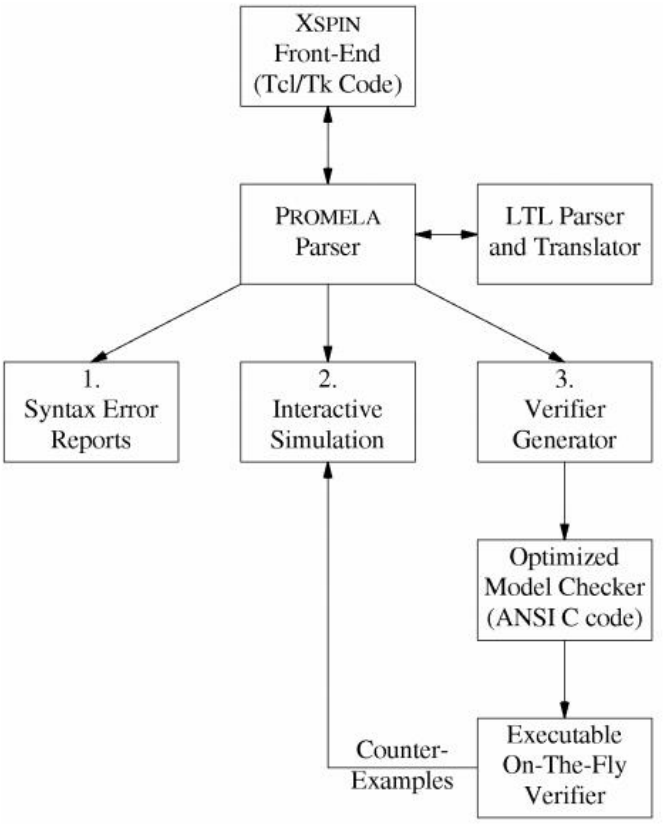

Figura 3-1 Estructura de simulación y verificación de SPIN. (Holzmann, 1997)

3.2. Un método para verificar propiedades en tiempo real de programas Ada

Thorsten Gerdsmeier y Rachel Cardell-Oliver del Departamento de ciencias de la computación de la Universidad de Essex proponen un método para poder verificar las propiedades en tiempo real de los programas Ada (Cardell-Oliver, 2001).

En la Figura 3-2, se observa la estructura de dicho método.

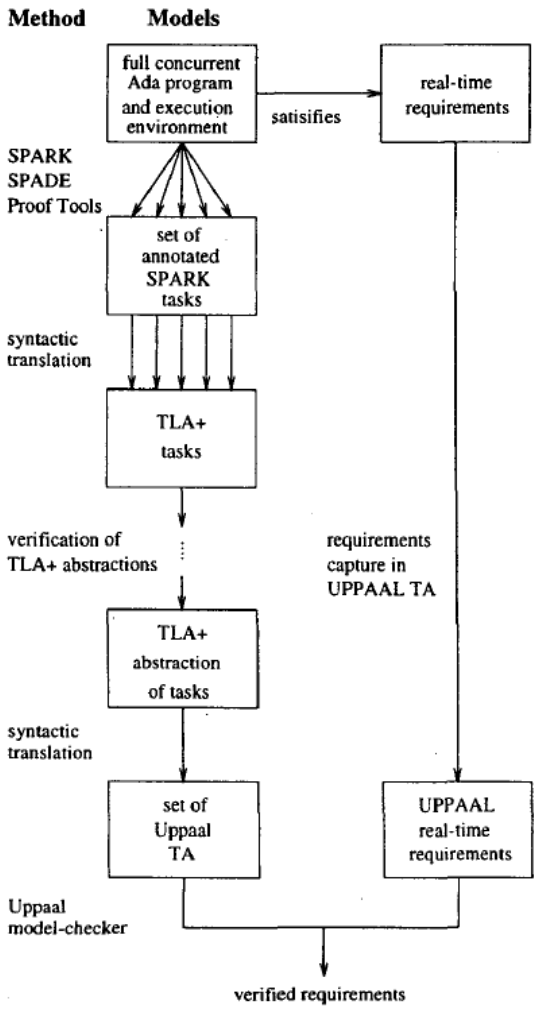

Figura 3-2 Estructura del Método. (Cardell-Oliver, 2001)

Su método consiste en utilizar la herramienta SPARK para hacer una traducción de las propiedades en tiempo real que deben ser verificadas en un programa Ada para pasarlos a unas acciones de lógica temporal. (TLA por sus siglas en inglés).

Una vez teniendo este TLA, se realiza una abstracción en conjunto del TLA y las especificaciones para crear un autómata que pueda ser representado en la herramienta Uppaal donde se realizará la verificación. Cabe mencionar que dicha transformación a lenguaje Uppaal no se realiza de manera automática: utiliza también un kernel con el protocolo del techo de prioridad inmediato (ICPP por sus siglas en inglés) para manejar la concurrencia del programa Ada.

3.3. RAST: Una herramienta para la extracción automática de modelos de programas Ada/SPARK

La facultad de ciencias de la Universidad de Porto, presenta una herramienta llamada RAST (Nuno Silva, 2011) mediante la cual es posible obtener de forma automática un modelo Uppaal a partir de un programa Ada/SPARK.

Básicamente la forma de trabajar de la herramienta se separa en 3 fases: procesamiento de los archivos de entrada, procesamiento de la información de los archivos fuente y traducción. Las cuales se detallarán a continuación:

Procesamiento de los archivos de entrada: Esta primera fase se enfoca fuertemente en transformar los archivos fuentes que recibe la herramienta, en archivos que contengan un formato más acorde para poder llevar acabo la transformación de los mismos. Para llevar a cabo dicha transformación se utilizan las siguientes aplicaciones:

- Avatox (Criley, 2006) aplicación que permite transformar algunas unidades de compilación de Ada en salidas ASIS representadas en un documento XML. Dicha representación XML representa comprensivamente al código fuente original.
- XALAN (The Apache Xml Project, 2019) aplicación para el procesamiento de archivos XML.

Procesamiento de la información de los archivos fuente: En esta fase se recolecta y almacena toda la información necesaria para poder llevar a cabo la transformación y creación de los programas Ada a modelos Uppaal. Con la información recolectada se crean diferentes gráficos de control de flujo los cuales representan toda la lógica de ejecución del programa original.

Traducción: En esta última fase, se realiza la traducción de todos los gráficos de control de flujo que se hayan generado en la fase anterior. Generando todos los elementos de Uppaal correspondientes a las construcciones de Ada usados en los programas fuente originales. En la Figura 3-3, se muestra el flujo de trabajo de la herramienta.

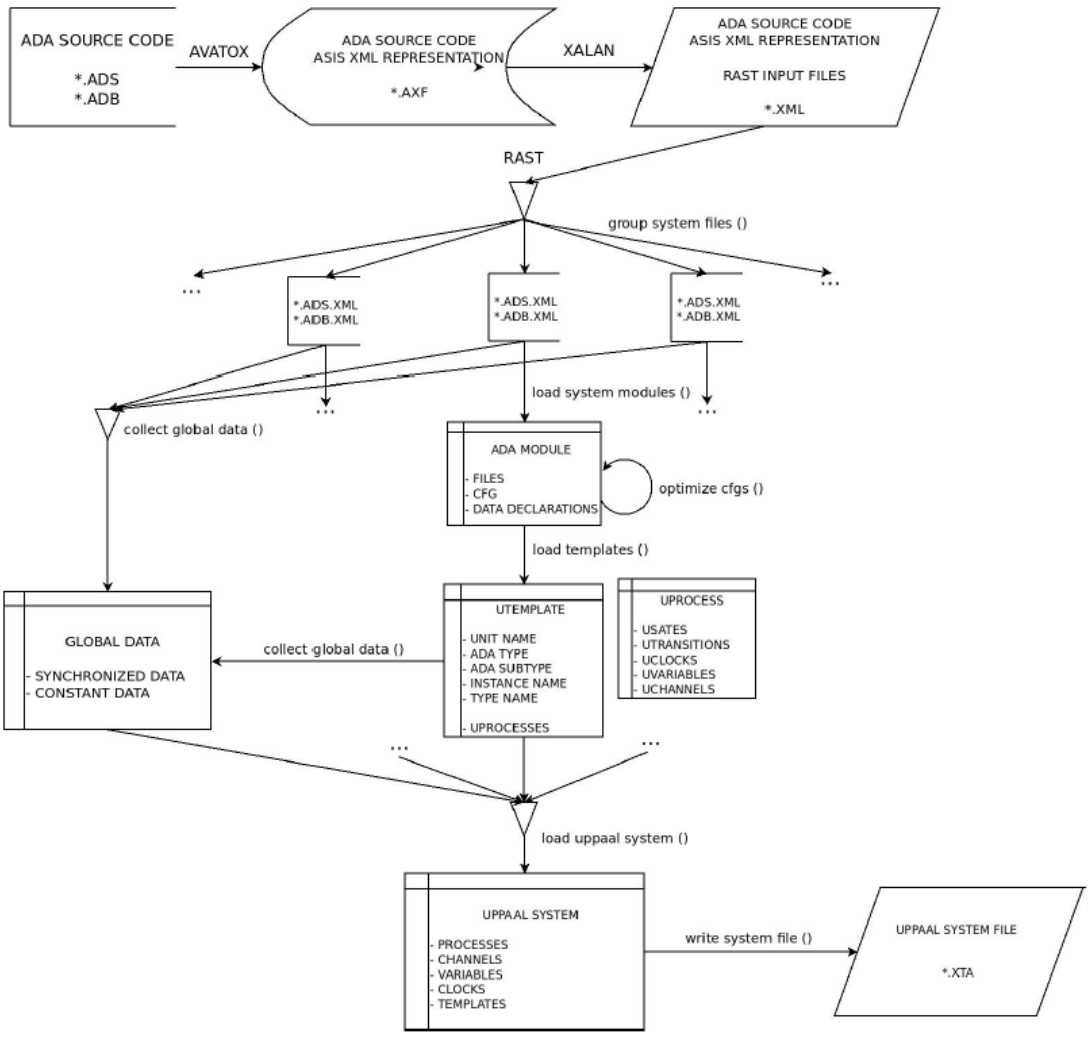

Figura 3-3 Flujo de trabajo RAST. (Nuno Silva, 2011)

La Tabla 3-1 muestra un resumen comparativo de las diferentes herramientas investigadas y la herramienta desarrollada para este trabajo de tesis UMAYA.

Herramienta/Método	Descripción	Ventaja	Desventaja
ATOS	Es una herramienta capaz de obtener modelos en SPIN de programas Ada en forma automática generándolos en su lenguaje de modelos PROMELA.	Cuenta con sus propios algoritmos de verificación de modelos y genera de forma automática sus modelos en PROMELA.	El lenguaje PROMELA es bastante complejo y limitado para poder obtener todas las características que un programa Ada pueda tener.
Método Gerdsmeier-Cardell	Método que traduce las propiedades de un programa Ada a TLA, las cuales se transforman a un modelo Uppaal.	Al obtenerse las TLA se generan modelos matemáticos junto con los requerimientos y las abstracciones, para después ser transformados a modelos Uppaal.	La traducción a modelos Uppaal no se genera de manera automática.
RAST	Es una herramienta mediante la cual es posible obtener de forma automática un modelo Uppaal a partir de un programa Ada/SPARK.	Es capaz de transformar una cantidad considerable de programas Ada a modelos Uppaal.	Tiene problemas para traducir programas grandes en tiempo real y que contienen módulos que interactúan entre sí.
UMAYA	Es una herramienta para transformar un programa Ada a modelo Uppaal.	Es posible transformar las construcciones básicas de un programa Uppaal.	No es posible transformar cualquier programa Ada que se presente.

Tabla *3*-1 *Tabla comparativa*

3.4. Cierre

En este capítulo se muestran diversos trabajos y herramientas, que intentaron también verificar las propiedades de los programas codificados en el lenguaje de programación Ada a través de diversas técnicas entre ellas la verificación de modelos. Cabe mencionar que la mayoría de estos trabajos no logró realizar la transformación automática del código Ada al lenguaje de modelo que decidieron utilizar. Y es ahí donde reside una de las aportaciones más valiosa de la herramienta UMAYA, al transformar automáticamente un programa Ada a modelo Uppaal.

Capítulo 4

4. Desarrollo del proyecto

En este capítulo se muestra el proceso de Ingeniería de Software seguido para el desarrollo de la herramienta UMAYA, presentando cada una de sus fases, actividades y entregables.

4.1. Modelo de Requisitos

Es necesario crear una herramienta que sea capaz de transformar un programa cuyo archivo fuente sea codificado en el lenguaje de programación Ada a un modelo Uppaal respetando la sintaxis y semántica del código fuente inicial.

La herramienta a realizar deberá cumplir con los requisitos funcionales mostrados en la Tabla 4-1.

Tabla *4*-1 Requisitos *funcionales*

ID	Requisito	Descripción
RF01	Transformar un archivo fuente Ada a XML	La herramienta deberá transformar el archivo fuente Ada proporcionado a un archivo XML, respetando la sintaxis y semántica del mismo.
RF02	Gestionar la información del XML	La herramienta deberá procesar la información del archivo XML generado para almacenarla e interpretarla, para posteriormente realizar la traducción a un modelo Uppaal.
RF03	Generar un archivo XML compatible con Uppaal	La herramienta generará un archivo XML que sea compatible con la herramienta Uppaal.

La Tabla 4-2 muestra los requisitos no funcionales tomados en cuenta para la elaboración del proyecto. Es importante se cumpla con el requisito de portabilidad para que la herramienta pueda ser ejecutada en cualquier ordenador que se requiera, cumpliendo con este requisito no funcional, esto se garantiza.

Tabla *4*-2 Requisitos *No funcionales*

ID	Calidad	Descripción
RC01	Portabilidad	La herramienta deberá de realizarse en el lenguaje de programación Java, ya que al ejecutarse el mismo en la máquina virtual de Java podrá ser portable a la mayoría de los sistemas operativos.

4.1.1. Restricciones

La tabla 4-3 muestra las restricciones consideradas en el proyecto. Tomando en cuenta las restricciones mostradas en la tabla, se evitarán problemas como la falta de compatibilidad entre las distintas versiones de Java.

Tabla 4-3 Restricciones

ID	Descripción
R01	La herramienta deberá ser desarrollada en Java 8. Ya que consideramos es una versión madura y estable.
R02	Solo se intentarán transformar archivos fuente de código Ada.

4.1.2. Caso de Uso

En esta sección se muestra el único Caso de uso que se lleva a cabo con la herramienta.

Tabla 4-4 Caso de Uso

ID	Descripción
CU01	Transformar un archivo de código Ada a un modelo Uppaal.

En la Figura 4-1 se muestra el diagrama de contexto en el cual el usuario interactúa con la herramienta UMAYA para realizar la transformación de un archivo Ada a un diagrama Uppaal.

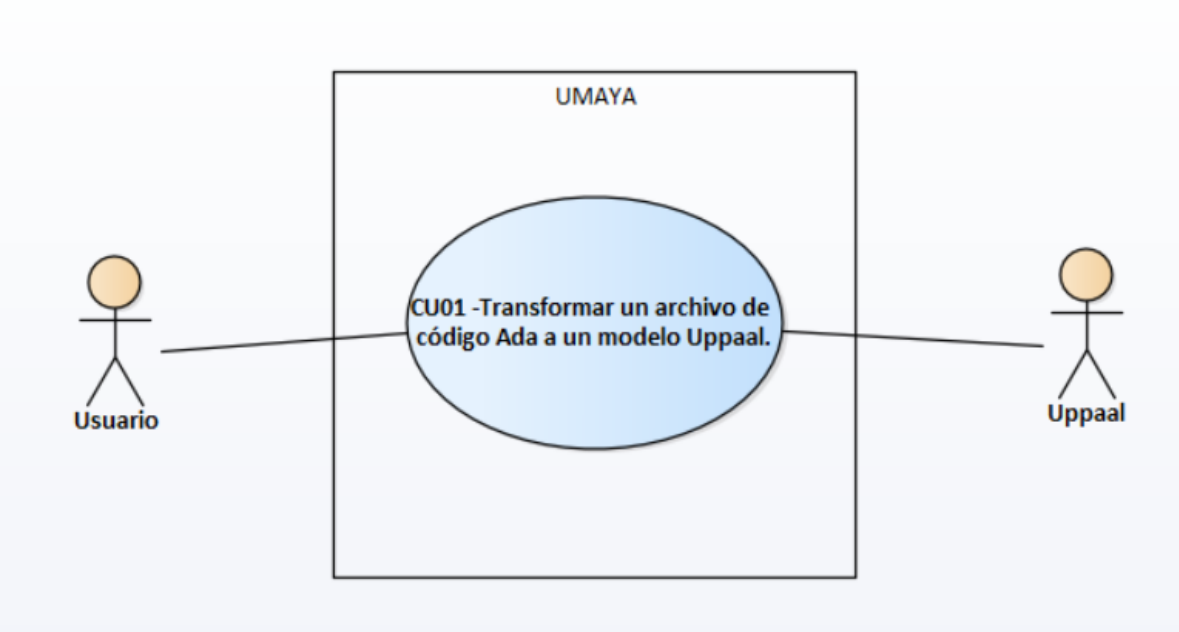

Figura 4-1 CU01.

La Figura 4-2 muestra la vista dinámica del diagrama de contexto.

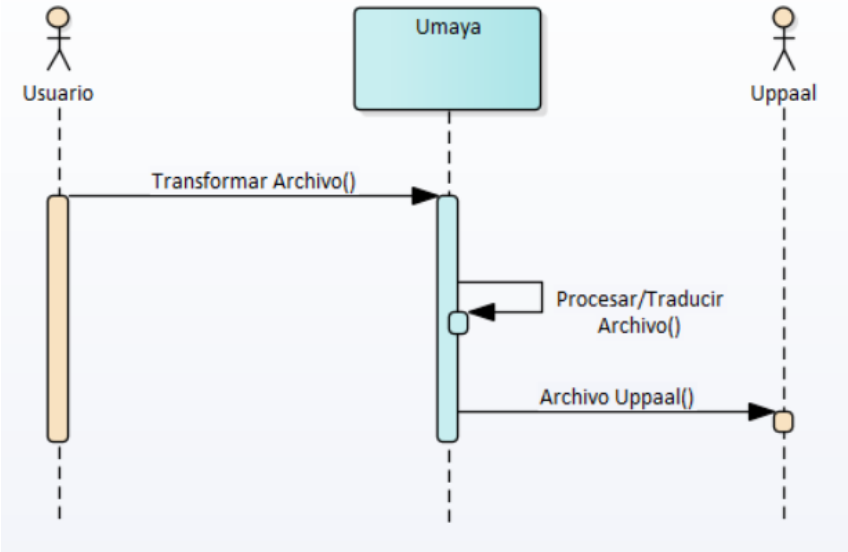

Figura 4-2 Vista dinámica del diagrama de contexto.

En ella se aprecia la interacción del usuario con la herramienta UMAYA. En primera instancia el usuario proporciona el archivo de código Ada, el cual posteriormente es analizado y procesado por la herramienta, para después generar un diagrama de modelo el cual pueda ser interpretado por Uppaal. El detalle de cada una de las fases del proceso es mostrado en la vista Arquitectónica.

4.2. Modelo de Diseño

En esta sección se presenta el modelo de diseño con sus diversas vistas.

4.2.1. Vista Arquitectónica

El proceso de transformación de un archivo en código Ada a un modelo Uppaal se divide en tres fases (Procesamiento de los archivos fuente Ada, Procesamiento de la información de los archivos fuente Ada y Traducción a un modelo Uppaal). La Figura 4-3 muestra el modelo arquitectónico utilizado, basado en el modelo de tuberías y filtros.

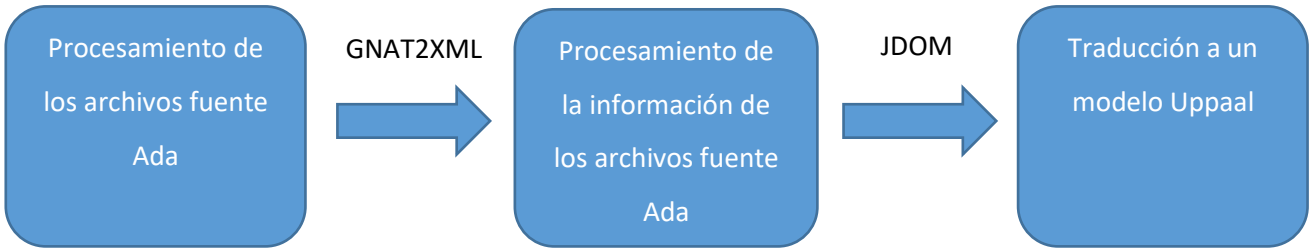

Figura 4-3 Modelo arquitectónico de la herramienta.

Procesamiento de los archivos fuente Ada

En esta fase se procesa el archivo fuente de código Ada, transformándolo en un archivo XML el cual es un poco más fácil de manipular. Para ello se utiliza la biblioteca GNAT2XML (belt, 2019) y la biblioteca JDOM (Hunter, 2019). La biblioteca GNAT2XML utiliza la especificación de la interfaz semántica de Ada (ASIS), la cual garantiza que al transformar el archivo de código fuente Ada al formato XML la sintaxis y la semántica del mismo quedan totalmente respetadas. La Figura 4-4 representa dicha fase.

Figura 4-4 Procesamiento de los archivos fuente Ada.

Procesamiento de la información de los archivos fuente Ada

En esta fase al utilizar la biblioteca JDOM para manipular la información contenida en el archivo XML creado por la biblioteca GNAT2XML se almacena y organiza la información necesaria para poder llevar a cabo la transformación y modelado del archivo de código fuente Ada a un diagrama de modelo Uppaal. Se comienza identificando las diferentes unidades de compilación que forman parte del archivo fuente (paquetes, procedimientos, funciones, etc). Una vez identificadas estas unidades de compilación, se crean los diferentes objetos Java que representan a dichas unidades en la herramienta UMAYA. Para así en dichos objetos almacenar, procesar y estructurar la información que se obtuvo de los XML creados anteriormente. Lo realizado en esta fase lo podemos observar en la Figura 4-5.

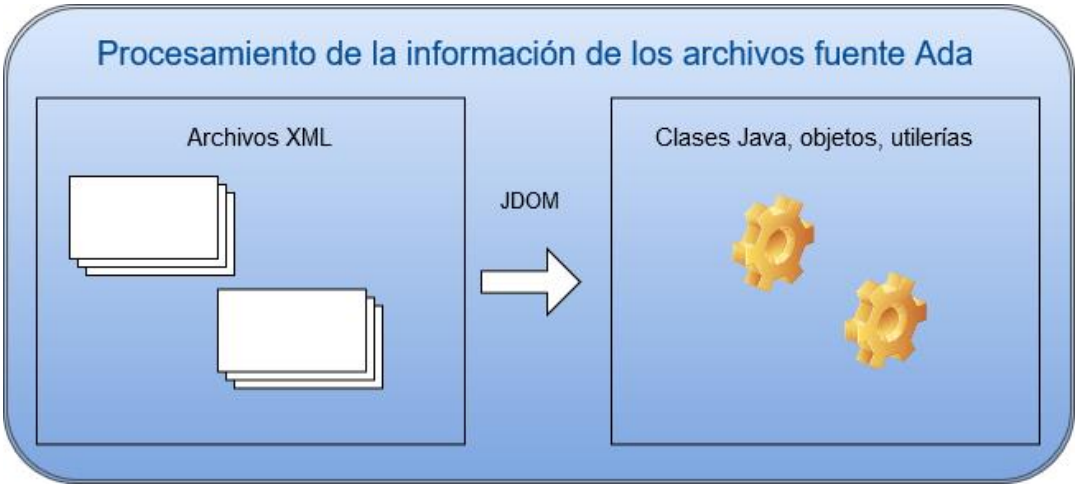

Figura 4-5 Procesamiento de la información de los archivos fuente Ada.

Traducción a un modelo Uppaal

La última fase del proceso es la traducción de las clases y objetos Java a los elementos equivalentes correspondientes en el lenguaje de modelado Uppaal. De acuerdo a los objetos Java que se han creado, se procede a crear los distintos *templates*, *locations*, *edges*, *guards*, etc., correspondientes a los mismos, dando lugar así al diagrama de modelo Uppaal. Cabe mencionar que para llevar a cabo dicha traducción se utiliza nuevamente la biblioteca JDOM para generar el nuevo XML que será reconocido por Uppaal como el diagrama de modelo. En la Figura 4-6 se muestra el diagrama de modelo Uppaal como resultado del XML generado por la herramienta.

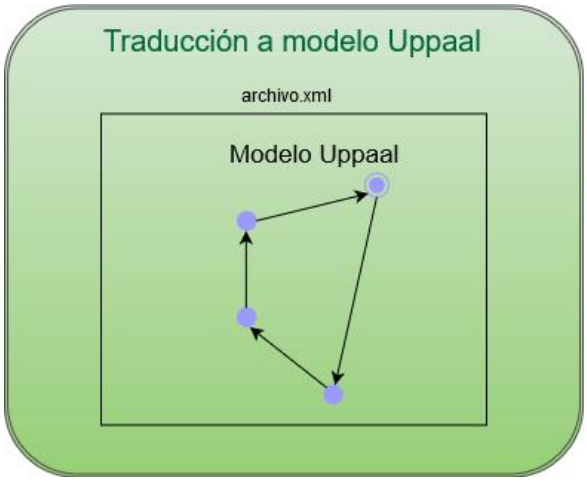

Figura 4-6 Traducción a modelo Uppaal.

4.2.2. Vista Estructural

En este apartado se mostrará la estructura de paquetes y clases que forman parte de la herramienta. La herramienta está conformada por cuatro paquetes principales, como se observa en la Figura 4-7.

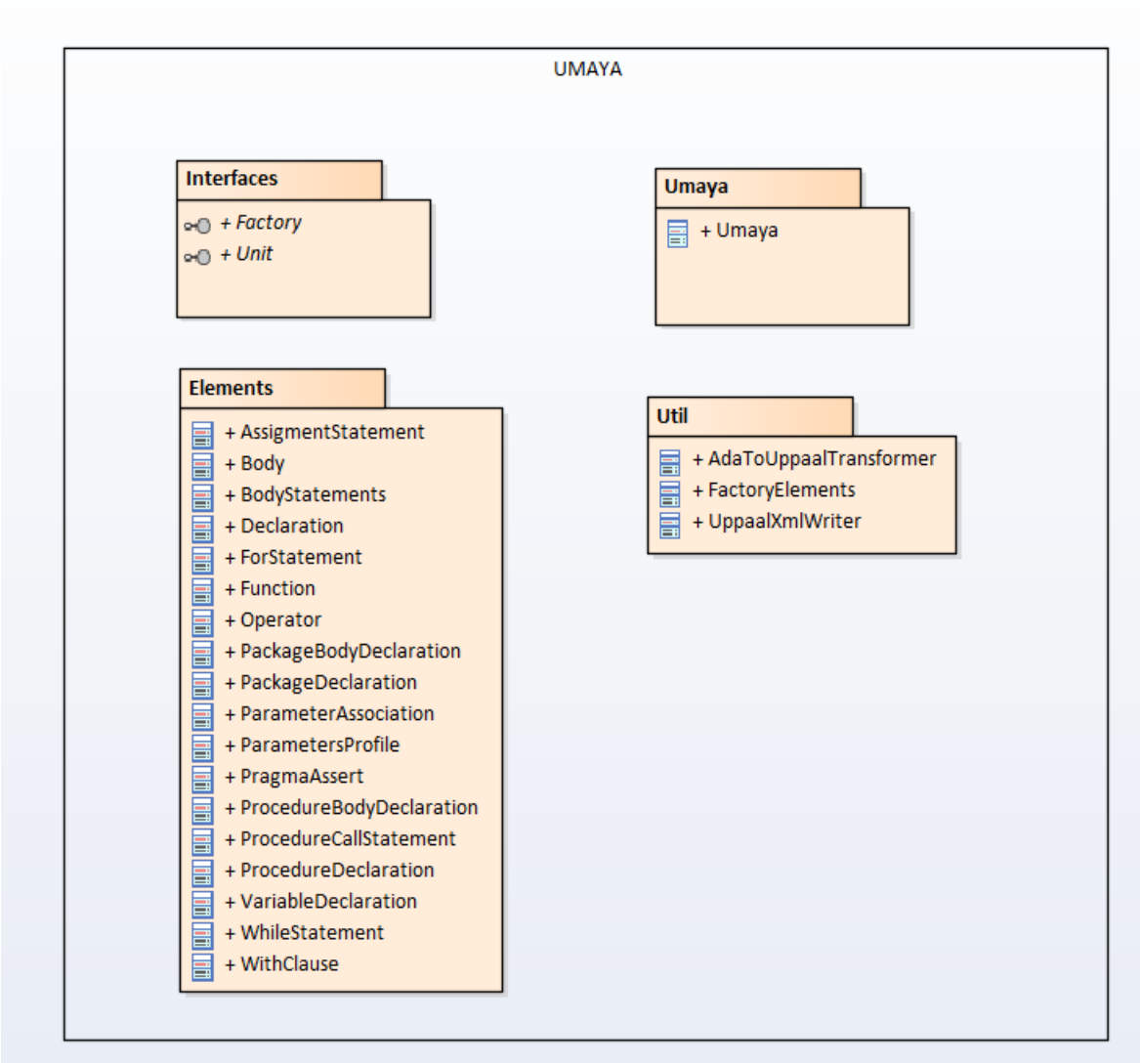

Figura 4-7 Paquetes UMAYA.

En el primer paquete, el paquete de interfaces, se encuentran las dos interfaces que se utilizan para crear los diferentes objetos necesarios para almacenar y procesar la información que se obtuvo de los archivos XML. En la Figura 4-8 se observan dichas interfaces.

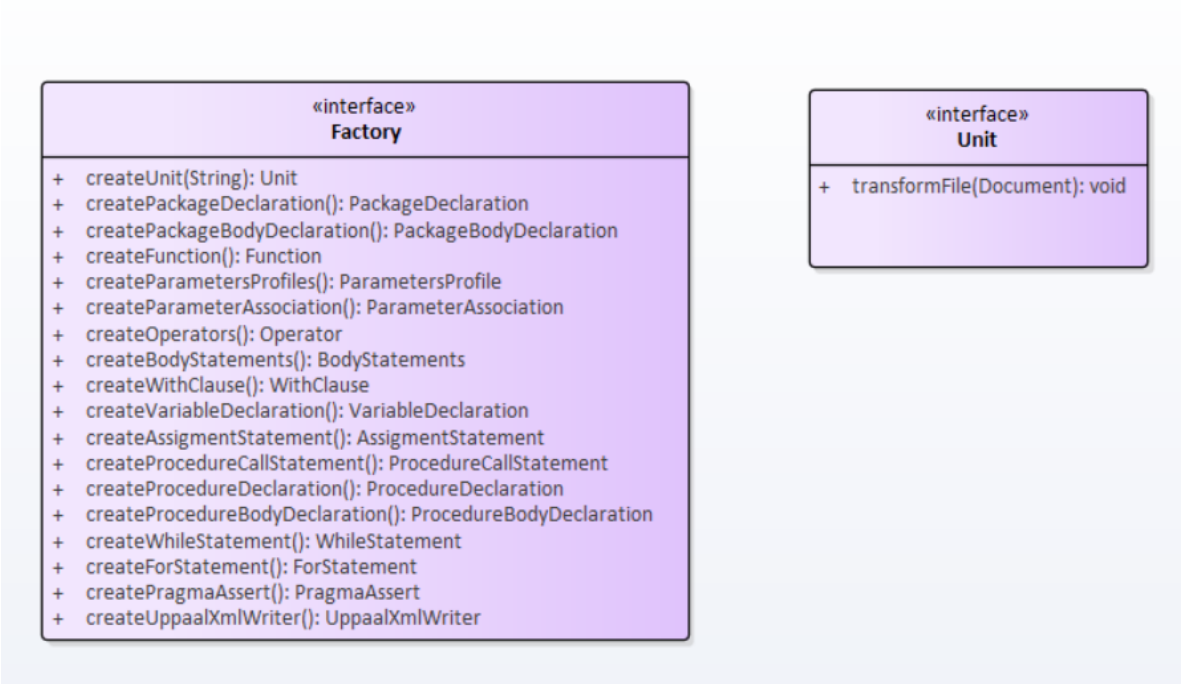

Figura 4-8 Interfaces.

La interfaz `Unit` es la abstracción que representa todas las posibles unidades de compilación que es posible encontrar al momento de analizar la información que proporciona el archivo XML al cual fue transformado el archivo fuente de código Ada. Estas pueden ser cualquier elemento que se encuentre en un programa como lo son la declaración de un paquete, el cuerpo de un paquete, la declaración de una función o procedimiento, el cuerpo de los mismos, etc.

Para la creación de los diferentes objetos que representan los distintos elementos que conforman un programa, se utiliza el patrón de diseño Factory (Eric Gamma, 1994). Para tener un mejor control y organización al momento de crear estos objetos. La abstracción que representa este patrón es la interfaz `Factory` mostrada en la Figura 4-8.

En el paquete `Elements` se encuentran todas las clases con las que se crean los objetos Java que representan a todos los posibles elementos de programación que se pueden encontrar

dentro de una unidad de compilación. Por estos elementos nos referimos a las diversas construcciones que se tomaron en cuenta para el proyecto como son las funciones, procedimientos, cláusulas with, ciclos while, for, etc. En la Figura 4-9, se muestran las dos clases principales de la herramienta UMAYA.

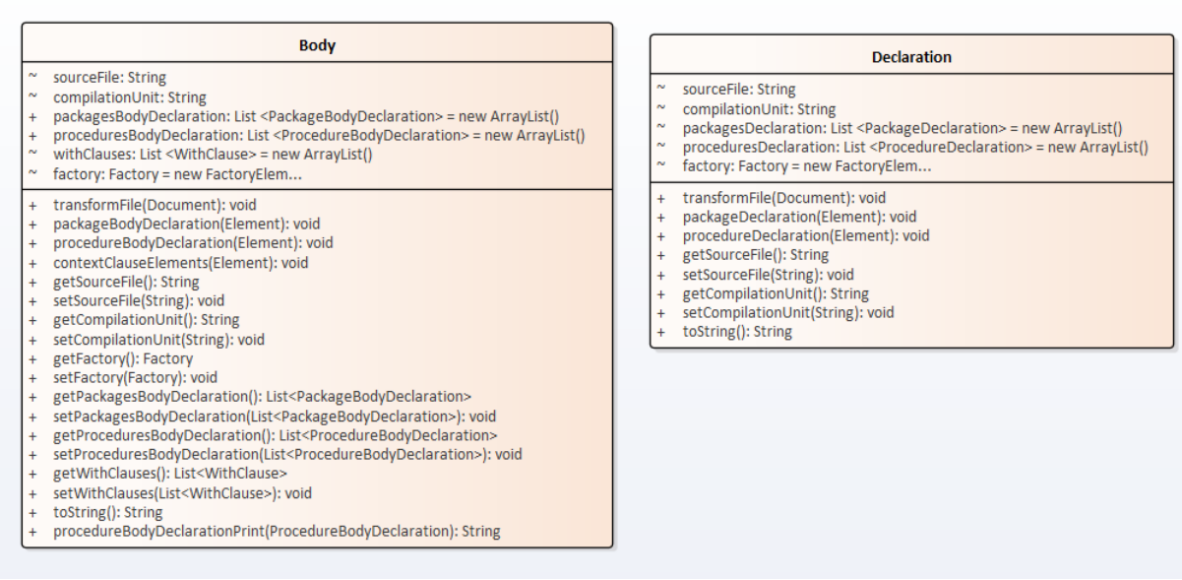

Figura 4-9 Clases principales.

En estas clases se observa que están compuestas por diversas listas de elementos que contienen a los objetos que representan a los diferentes elementos que pueden conformar un programa. Tomando por ejemplo un procedimiento, se sabe que el cuerpo de un programa en Ada puede tener de 1 a N procedimientos. Es por eso que en la clase `Body` se cuenta con una lista de procedimientos donde se van almacenando cada uno de los procedimientos con los que cuente el programa.

A su vez, cada objeto de tipo procedimiento contiene los objetos correspondientes que representan a cada elemento que puede conformar un procedimiento. Así, dependiendo de la estructura que tenga el programa, estos objetos se van almacenando unos dentro de otros, manteniendo la jerarquía entre ellos y respetando la semántica del programa conforme se va obteniendo la información del XML sintáctica y semánticamente formado.

El paquete `Umaya` contiene solo la clase main encargada de ejecutar la funcionalidad de la herramienta completa, mediante el método `transformFile()` como se observa en la Figura 4-10.

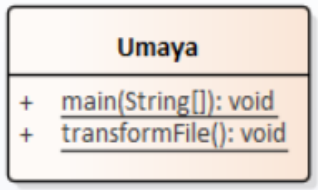

Figura 4-10 Clase main.

Por último, está el paquete `Util`, en el cual se encuentran las clases de utilería encargadas de realizar las diversas acciones que ejecuta nuestra herramienta para llevar acabo la transformación de un programa Ada a un diagrama de modelo Uppaal. La clase `AdaToUppaalTransformer` contiene los métodos para transformar el archivo fuente Ada al XML ya mencionado que se genera en la primera fase del proceso de transformación. Contiene también el método para analizar dicho XML y comenzar con la creación de los diversos objetos Java que contienen toda la información referente al programa Ada para poder hacer la traducción al diagrama de modelo Uppaal. Cuenta también con el método que realiza dicha traducción y genera el XML con un formato reconocible para Uppaal.

La clase `FactoryElements` es la especificación de la interfaz Factory, la cual es la encargada de crear todos los objetos que representan todos los elementos que conforman el programa.

Al final, la clase `UppaalXmlWriter` es quien realiza la traducción de toda la información almacenada y procesada en nuestros objetos Java, a un formato XML que Uppaal pueda reconocer. En la Figura 4-11, se observan las tres clases mencionadas.

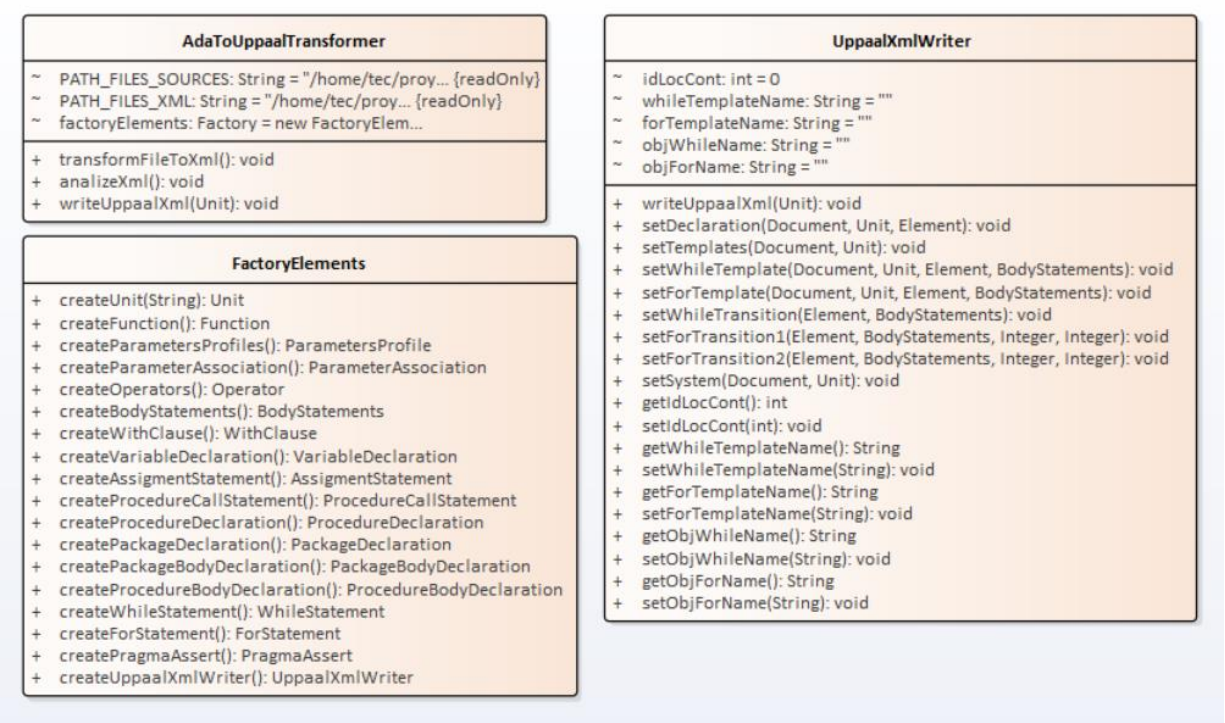

Figura 4-11 Clases Utilería.

4.2.3. Vista Dinámica

La Figura 4-12 muestra la Vista Dinámica del proceso de transformación llevado a cabo por la herramienta UMAYA.

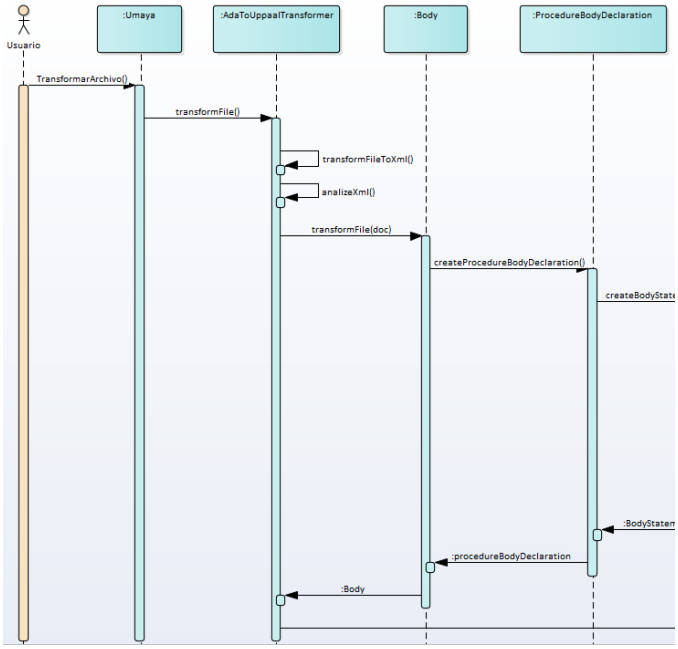

Figura 4-12 Vista Dinámica primera parte.

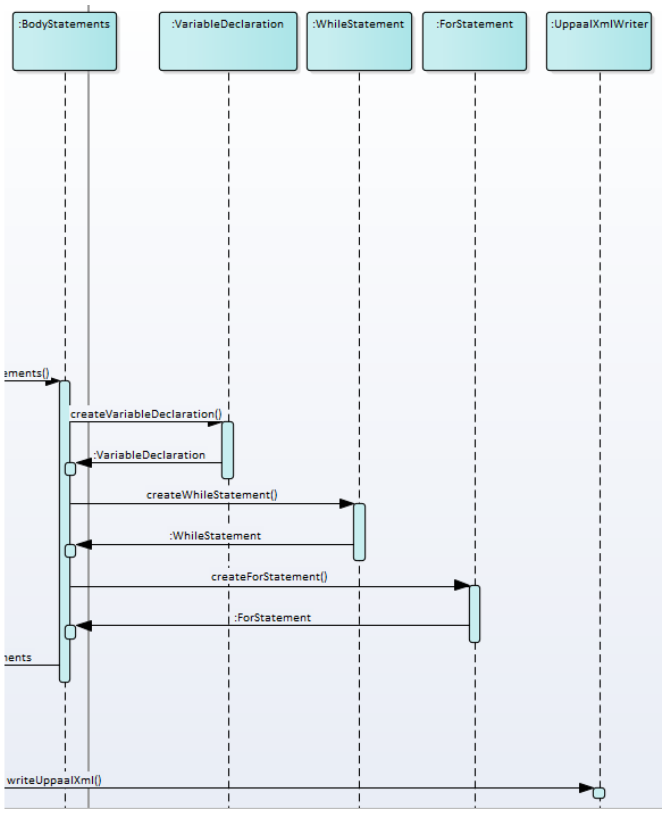

Figura 4-13 Vista Dinámica segunda parte.

En la Figura 4-12 y Figura 4-13 se aprecia de manera general un ejemplo de las clases implicadas en el proceso de transformación realizado por la herramienta UMAYA al momento de transformar un archivo de código Ada a un diagrama de modelo Uppaal. La clase `AdaToUppaalTransformer` es quien inicia el proceso de transformación a través del método `transformFileToXml()`, en el cual el archivo de código Ada es transformado a un archivo XML que contiene toda la información sintáctica y semántica del programa a transformar. Ya que se cuenta con dicho archivo, este es analizado y procesado, para ello la herramienta utiliza las clases `Body`, `ProcedureBodyDeclaration`, `BodyStatements`, `VariableDeclaration`, `WhileStatement` y `ForStatement`. Al final toda la información recabada del programa a transformar en los objetos Java creados por la herramienta, es traducida a un diagrama de modelo Uppaal por la clase `UppaalXmlWriter`.

4.3. Modelo de Implementación

En esta sección se mostrará la codificación de algunos de los métodos utilizados en la herramienta, como ejemplo del modelo de Implementación.

En la Figura 4-13 se observa el método `transformFileToXML()` que es el encargado de convertir los programas Ada a XML utilizando la biblioteca GNAT2XML.

```
public void transformFileToXml() {

    try (Stream <Path> paths = Files.walk(Paths.get(this.PATH_FILES_SOURCES))) {
        paths.filter(Files::isRegularFile)
             .forEach(file ->{
                 try {
                    String linuxCommand;
                    Process executeProcess;
                    linuxCommand= "gnat2xml -v  -m"+ this.PATH_FILES_SOURCES + "/xmlAdaPrograms "
                    + this.PATH_FILES_SOURCES + "/" + file.getFileName().toString()+ " -cargs -gnat2012";
                    executeProcess= Runtime.getRuntime().exec(linuxCommand);
                    BufferedReader consoleMessage = new BufferedReader(new InputStreamReader(executeProcess.getInputStream()));
                    executeProcess.waitFor();
                    consoleMessage.lines().forEach(line -> {System.out.println(line);});

                 } catch (IOException | InterruptedException e) {
                     // TODO Auto-generated catch block
                     e.printStackTrace();
                 }});
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Figura 4-14 Método transformFileToXml.

Como se muestra en la Figura 4-14, el método `transformFileToXML()` toma de la ruta definida por la constante PATH_FILES_SOURCES, el archivo de código Ada que mediante una instrucción del sistema operativo Linux, se transforma utilizando la biblioteca GNAT2XML en un archivo XML más fácil de interpretar y que respeta tanto la sintaxis como la semántica del programa original. Este XML será utilizado después para obtener la información necesaria para lograr la transformación del programa original Ada a un diagrama de modelo Uppaal.

La Figura 4-15 muestra el método que comienza con el análisis del archivo XML generado por GNAT2XML, el cual es llamado `analizeXml()`.

```
public void analizeXml(){
        try (Stream <Path> paths = Files.walk(Paths.get(this.PATH_FILES_XML))) {
        paths.filter(Files::isRegularFile)
             .forEach(file ->
                      {
                     try {
                         String pathFileToAnalize = this.PATH_FILES_XML +"/"+ file.getFileName().toString();
                         String sourceFile;
                         SAXBuilder builder = new SAXBuilder();
                         Document doc = builder.build(pathFileToAnalize);
                         Element root = doc.getRootElement();
                         sourceFile = root.getAttributeValue("source_file");
                         String[] splitExtension = sourceFile.split("\\.");
                         String extensionFile = splitExtension[1];
                         Unit unit = factoryElements.createUnit(extensionFile);
                         unit.transformFile(doc);
                         if(unit.getClass().getSimpleName().compareTo("Body")==0){
                             this.writeUppaalXml(unit);
                             System.out.println("El archivo se creo correctamente");
                         }
                     } catch (JDOMException ex) {
                         Logger.getLogger(AdaToUppaalTransformer.class.getName()).log(Level.SEVERE, null, ex);
                     } catch (IOException ex) {
                         Logger.getLogger(AdaToUppaalTransformer.class.getName()).log(Level.SEVERE, null, ex);
                     }
                     });
    } catch (IOException e) {
        // TODO Auto-generated catch block
        e.printStackTrace();
    }
}
```

Figura 4-15 Método analizeXml.

En el método `analizeXml()` se toma el archivo XML generado que se encuentra en la ruta definida por la constante PATH_FILES_XML y mediante la biblioteca JDOM se genera un documento que contiene toda información de dicho archivo en forma de nodos acomodados jerárquicamente. Dependiendo del tipo de unidad de compilación que se detecte al analizar el archivo XML, se crea un objeto `Unit` que la representa, en este caso la unidad será un

objeto del tipo `Body`. Teniendo la unidad de compilación ya definida, se comienza con la transformación del archivo a un diagrama de modelo Uppaal.

Por último, en la Figura 4-16 se muestra el método `transformFile()`, el cual comienza con la creación de los diversos objetos Java utilizados para la recopilación de la información del programa Ada codificado.

```
public void transformFile(Document doc) {

    Element root = doc.getRootElement();
    Element nodeChild;
    this.setSourceFile(root.getAttributeValue("source_file"));
    this.setCompilationUnit(root.getAttributeValue("unit_kind"));

    if(compilationUnit.compareTo("A_Package_Body")==0){
        this.packageBodyDeclaration(root);
    }

    if(compilationUnit.compareTo("A_Procedure_Body")==0){
        nodeChild = root.getChild("context_clause_elements_ql");
        this.contextClauseElements(nodeChild);
        nodeChild = root.getChild("unit_declaration_q");
        nodeChild = nodeChild.getChild("procedure_body_declaration");
        this.procedureBodyDeclaration(nodeChild);
    }
}
```

Figura 4-16 Método transformFile.

En el método `transformFile()` se recibe como parámetro el documento anteriormente creado en el método `analizeXml()`, se analiza cada uno de sus nodos y al detectar los diferentes elementos que componen el programa, se procede a la creación de los distintos objetos que representarán y mantendrán la información de cada uno de ellos.

Estos son solo algunos de los múltiples métodos utilizados en la herramienta UMAYA. Si se desea analizar el código completo de la herramienta, se puede descargar del siguiente repositorio de gitlab: https://gitlab.com/iscfcohdez/proyectoumaya.git

4.4. Cierre

En este capítulo se mostró el proceso de Ingeniería de Software llevado a cabo para la realización de la herramienta UMAYA. Se presentaron los diversos modelos utilizados, desde el modelo de requisitos, diseño e implementación. Así como las distintas vistas utilizadas (arquitectónica, estructural y dinámica) al momento de diseñar la herramienta, las actividades realizadas y entregables siguiendo un enfoque ingenieril en el desarrollo del software.

Capítulo 5

5. Pruebas

En este capítulo se explicará la manera en que se realizó la prueba que valida la hipótesis propuesta para este trabajo de tesis. Esta prueba se realizó mediante una prueba de concepto que demuestra la utilidad de la hipótesis y los beneficios que se pueden obtener al explotarse la misma.

5.1. Prueba de Concepto

La prueba de concepto consistirá en la transformación de un programa de código Ada a un diagrama de modelo Uppaal, pasando por todas las fases del proceso explicadas en el capítulo anterior.

5.1.1. Procesamiento de los archivos fuente Ada

Para la realización de la prueba de concepto se utilizó el código Ada que se muestra en la Figura 5-1.

```
package body array7 with
  SPARK_Mode => on
is


   procedure Main is
      N: Integer:= 100000;
      src: array (0..N) of Integer;
      dst: array (0..N) of Integer;
      i: Integer:=0;
   begin

      while (src(i) /= 0) loop
         dst(i) := src(i);
         i := i + 1;
      end loop;

      for  x in Integer range 0..i-1 loop
         pragma Assert(dst(x) = src(x));
      end loop;

   end Main;


end array7;
```

Figura 5-1 Código fuente Ada.

El programa está conformado de un procedimiento que contiene un ciclo `While` y un ciclo `For`. El ciclo `While` lo utiliza para copiar los valores de un arreglo de dimensión `N` a otro arreglo del mismo tamaño hasta que uno de sus valores sea igual a 0. Mientras que el ciclo `For` se usa para verificar que el copiado de información se realice correctamente.

Como se mencionó anteriormente, este código fuente Ada mediante la biblioteca GNAT2XML es transformado en un archivo XML que se analizará para obtener la información necesaria para realizar la transformación a un diagrama de modelo Uppaal.

5.1.2. Procesamiento de la información de los archivos fuente Ada

Una vez transformado el código fuente y empezado el análisis del archivo XML creado, el primer elemento encontrado es una unidad de compilación el cual es el cuerpo de un programa. Por lo que se utiliza la clase `Body` para empezar a recolectar la información del programa. Ya que se identifica esto, el primer objeto creado será la instanciación de la clase `Body`, la cual contendrá otro objeto de tipo procedimiento como lo muestra la Figura 5-2.

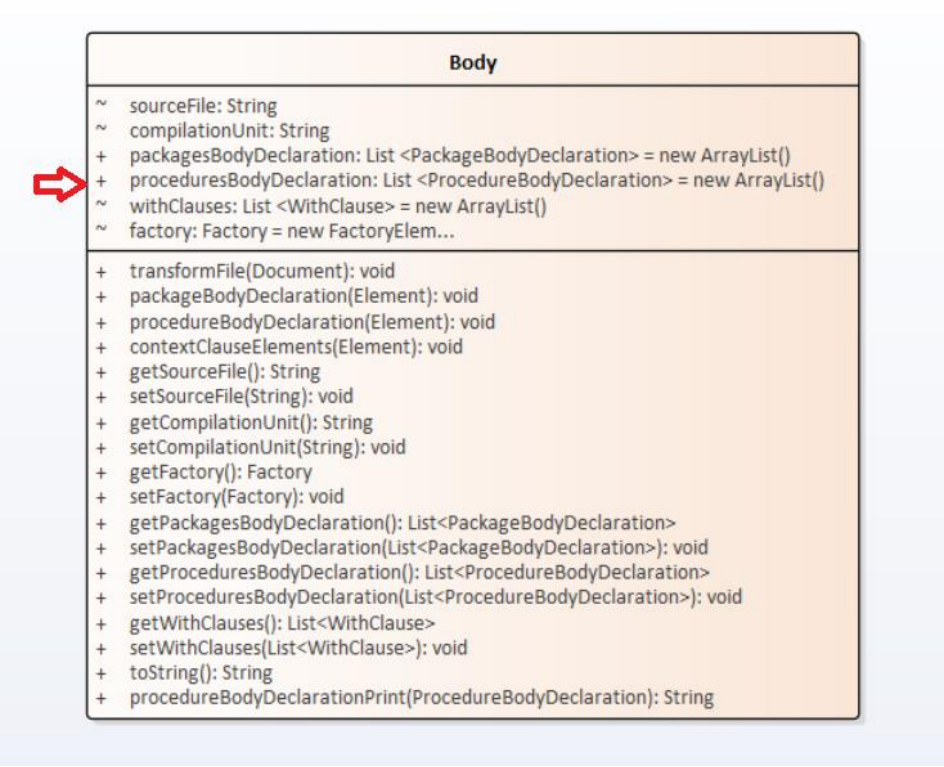

Figura 5-2 Clase Body.

Para almacenar toda la información contenida en el procedimiento que contiene el programa, se utilizará la clase `ProcedureBodyDeclaration`, la cual estará contenida dentro de la clase `Body` como se mencionó anteriormente. Dicha clase contiene a su vez la clase `BodyStatements` la cual almacena la información de todos los posibles elementos que contenga un procedimiento como puede ser un ciclo `While` y un ciclo `For` como es el caso. En la Figura 5-3 podemos observar dichas clases y la relación entre las mismas.

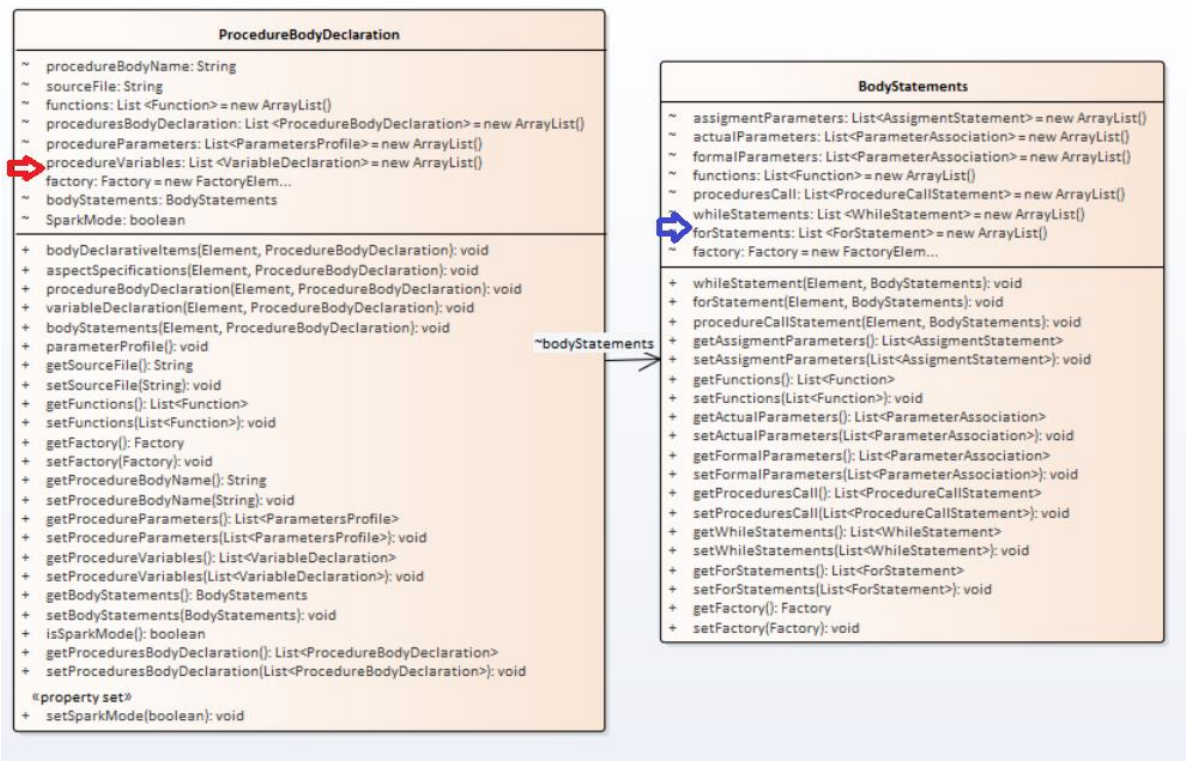

Figura 5-3 Clases ProcedureBodyDeclaration y BodyStatements.

En la Figura 5-3 se aprecia como la clase `BodyStatements` contiene dos objetos del tipo `WhileStatement` y `ForStatement` con toda la información referente a los 2 ciclos utilizados en el programa. A su vez, la clase `ProcedureBodyDeclaration` contiene una lista de objetos de tipo `VariableDeclaration` donde se maneja la información de la declaración de variables realizada en el procedimiento. En la Figura 5-4 se observa dicha clase.

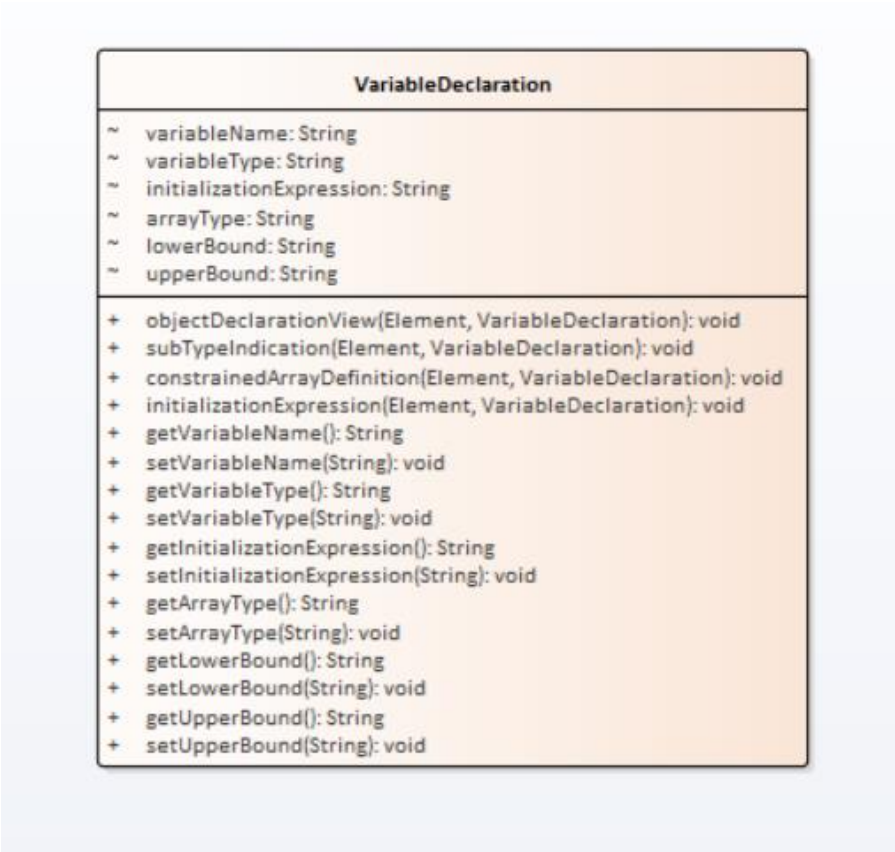

Figura 5-4 Clase VariableDeclaration.

Las clases `WhileStatement`, `ForStatement` y `Function` se muestran en la Figura 5-5.

Figura 5-5 Clases Function,WhileStatement y ForStatement.

La clase `Function` se encuentra relacionada con la clase `WhileStatement` y `ForStatement` ya que al transformar el archivo de código fuente Ada a XML la biblioteca identifica la condición del ciclo `While` y la definición de rango de valores del ciclo `For` como una función. Es por ello que cada uno de los ciclos contiene a su vez un objeto de tipo `Function` para definir estas dos expresiones. Cabe mencionar también que ese objeto `Function` se utiliza de forma general para cualquier tipo de función que se encuentre en el programa.

5.1.3. Traducción a un modelo Uppaal

Después de la ejecución de diversos métodos y procesamiento de los datos en los diversos objetos Java que se han creado, es la clase `UppaalXmlWriter` la que realiza la traducción de toda esta información a un archivo XML el cual pueda ser interpretado por Uppaal. En la Figura 5-6 se muestra dicha clase.

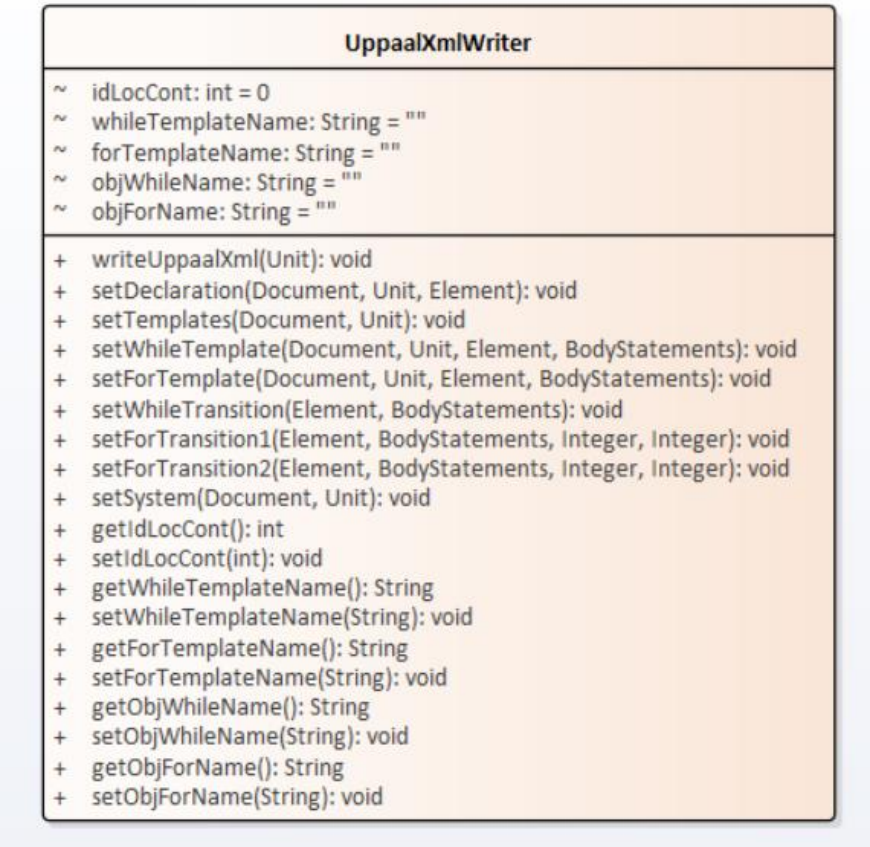

Figura 5-6 Clase UppaalXmlWriter.

Al final del proceso realizado por la herramienta UMAYA, se obtiene como resultado el siguiente XML mostrado en la Figura 5-7.

```
<?xml version="1.0" encoding="UTF-8"?>
<nta>
  <declaration>const int N:=100000;
int [0,N] src[N] ;
int [0,N] dst[N] ;
int i:=0;
int x:=0;</declaration>
  <template>
    <name x="5" y="5">WhileLoop</name>
    <declaration />
    <location id="id0" x="-704" y="-280">
      <name x="-760" y="-320">while</name>
    </location>
    <init ref="id0" />
    <transition>
      <source ref="id0" />
      <target ref="id0" />
      <label kind="guard" x="-632" y="-392">src[i]!=0</label>
      <label kind="assignment" x="-632" y="-208">dst[i]:=src[i],
i:=i+1</label>
      <nail x="-704" y="-408" />
      <nail x="-504" y="-408" />
      <nail x="-496" y="-160" />
      <nail x="-704" y="-160" />
      <nail x="-704" y="-264" />
    </transition>
  </template>
  <template>
    <name>ForLoop</name>
    <declaration />
    <location id="id1" x="-120" y="-112" />
    <location id="id2" x="-336" y="-112">
      <name x="-376" y="-120">for</name>
    </location>
    <init ref="id2" />
    <transition>
      <source ref="id1" />
      <target ref="id2" />
      <label kind="guard" x="-272" y="-8">dst[x]==src[x]</label>
      <nail x="-120" y="-24" />
      <nail x="-336" y="-24" />
    </transition>
    <transition>
      <source ref="id2" />
      <target ref="id1" />
      <label kind="guard" x="-264" y="-208">x&lt;(i-1)</label>
      <nail x="-336" y="-216" />
      <nail x="-120" y="-216" />
    </transition>
  </template>
  <system>whileLoop=WhileLoop();
forLoop=ForLoop();
system whileLoop, forLoop;</system>
</nta>
```

Figura 5-7 XML formato Uppaal.

En este nuevo XML generado, se pueden observar los diversos elementos correspondientes en Uppaal a los elementos que contiene el programa original. En primera instancia se encuentra la sección de declaración de variables, muy parecida a la sección donde se declaran las variables en el programa y que contiene las variables necesarias para la ejecución del mismo.

Después está conformado por dos *templates* las cuales corresponden al ciclo `While` y `For` respectivamente. Cada uno de estos *templates* está formado por *locations* y *transitions* que son los elementos encargados de simular los dos ciclos del programa original.

En el *template* que representa el ciclo `While`, tenemos un location que representa el estado inicial del ciclo, seguida de una *transition* que realiza la ejecución del ciclo utilizando un *guard* para verificar la condición del ciclo y un elemento *assignment* para realizar las dos asignaciones que se realizan dentro de este.

El *template* del ciclo `For`, cuenta con dos *transitions*, dos *locations* y dos *guards*. La primera transición es para verificar el rango de valores con el cual se ejecutará el ciclo `For` y la segunda para verificar la aserción de que cada uno de los valores de los arreglos son iguales.

Al final el diagrama de modelo Uppaal quedo de la siguiente manera:

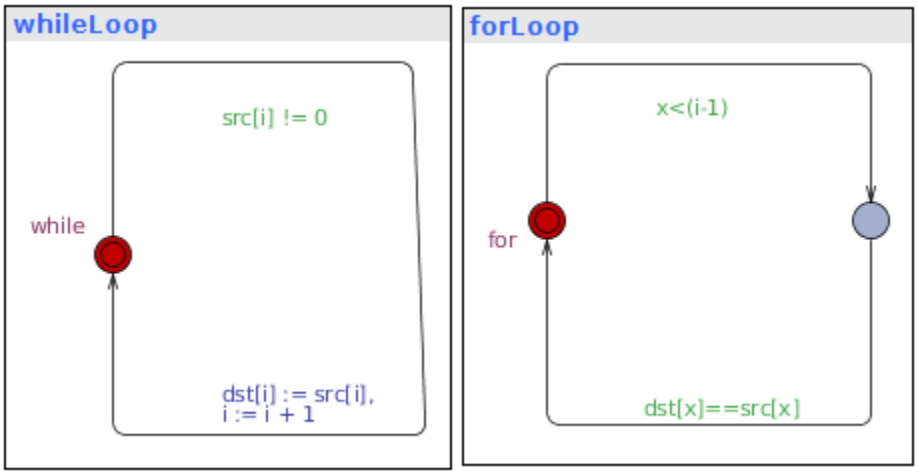

Figura 5-8 Diagrama de Modelo Uppaal.

5.2. Cierre

En este capítulo se describió la prueba realizada, pasando a través de las distintas fases de trabajo de la herramienta UMAYA. Logrando así la correcta transformación de un programa de código Ada a un diagrama de modelo Uppaal, comprobando de esta manera la hipótesis presentada para este trabajo de tesis.

Capítulo 6

6. Conclusiones y trabajo futuro

En este capítulo exponemos la conclusión a la que se llegó en este trabajo de investigación y el posible trabajo a futuro que podría enriquecer al mismo.

6.1. Conclusiones

Es posible transformar automáticamente un programa fuente de código Ada a un diagrama de modelo Uppaal. El ejemplo que se utilizó para la demostración de la hipótesis, fue solo una prueba de concepto con la cual se demostró que es posible realizar dicha transformación planteada. Para así facilitar a los desarrolladores de programas Ada, la posibilidad de aplicar técnicas de verificación como la verificación de modelos.

La herramienta UMAYA aún se encuentra en una etapa inicial con la que es posible transformar algunas de las construcciones más utilizados en la elaboración de programas Ada. Aún falta para poder tener una herramienta lo suficientemente madura que pueda enfrentar cualquier escenario de programación en código Ada que se le presente. Sin embargo, fue posible realizar nuestra prueba de concepto y comprobar nuestra hipótesis.

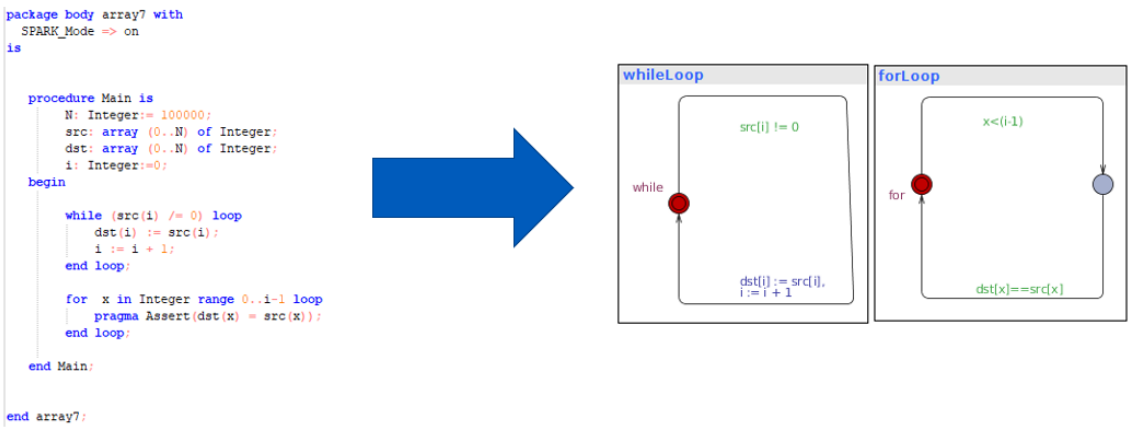

Figura 6-1 Transformación Ada a Uppaal.

6.2. Trabajo futuro

El proyecto tiene potencial para ser ampliado y mejorado, de manera que UMAYA logre ser una herramienta bastante útil para ayudar en la traducción de programas Ada a diagramas de modelo Uppaal. En un futuro podrían agregarse más construcciones para ser tomados en cuenta en el proceso de transformación, ya que la herramienta está preparada para ser extensible en cualquier momento.

Podría mejorarse también el algoritmo para el manejo y procesamiento de la información obtenida de los archivos XML, implementando patrones de diseños como el patrón visitor. Por último, podría agregarse el manejo de múltiples archivos a transformar, ya que en Ada es muy común que los programas se dividan en archivos .ads y .adb.

Bibliografía

1994-2019 The MathWorks, I. (09 de Mayo de 2019). *PolySpace*. Obtenido de https://www.mathworks.com/products/polyspace.html

AdaCore, C. ©. (09 de Mayo de 2019). *GNATCheck*. Obtenido de https://www.adacore.com/gnatpro/toolsuite/gnatcheck

Adalog, 2. (09 de Mayo de 2019). *AdaControl*. Obtenido de http://www.adalog.fr/en/adacontrol.html

Barnes, J. (2014). *Programming in Ada 2012.* Cambridge University Press; 1 edition.

Barnes, J. G. (2003). *High Integrity Software - The SPARK Approach to Safety and Security.* Addison-Wesley.

belt, j. (29 de 03 de 2019). *GNAT2XML git project*. Obtenido de https://github.com/sireum/archived-v2-bakar-gnat2xml

BULL computers chronological history. (24 de 05 de 2019). Obtenido de http://www.feb-patrimoine.com/histoire/english/chronoa9.htm

Cardell-Oliver, T. G. (2001). Proceedings Seventh IEEE International Conference on Engineering of Complex Computer Systems. *A Method for Verifying Real-Time Properties of Ada Programs.* Skovde, Sweden, Sweden: IEEE.

Criley, M. A. (2006). Ada user journal. *Avatox — Ada to XML.*

Dill, R. A. (1994). A Theory of Timed Automata. *TCS*, 183-235.

Eric Gamma, R. H. (1994). *Design Patterns.* Addison Wesley.

Faria J.M., M. J. (2012). An Approach to Model Checking Ada Programs. *Reliable Software Technologies – Ada-Europe 2012.* (págs. 105-118). Brorsson M: Springer, Berlin, Heidelberg.

G. Behrmann, A. D. (2004). A Tutorial on Uppaal. *In SFM, volume 3185 of LNCS* (págs. 200-236). Springer.

Gang Tan, P. S. (25 de April de 2019). *A Collection of Well-Known Software Failures*. Obtenido de http://www.cse.psu.edu/~gxt29/bug/softwarebug.html

Holzmann, G. J. (1997). The Model Checker SPIN. *IEEE TRANSACTIONS ON SOFTWARE ENGINEERING, VOL. 23, NO. 5.*

Hunter, J. (29 de 03 de 2019). *JDOM Homepage*. Obtenido de http://www.jdom.org/

ISO. (30 de 05 de 2019). *International Organization for Standardization*. Obtenido de https://www.iso.org/home.html

Katoen, C. B.-P. (2008). *Principles of Model Checking.* MIT Press.

Merz, S. (2001). Model checking: a tutorial overview. *Springer-Verlag New York, Inc., New York, NY, USA*, pág. 338.

Meyer, B. (1997). *Object-Oriented Software Construction 2nd Edition.* Prentice Hall; 2 edition.

Nuno Silva, N. M. (2011). *A Tool for Automatic Model Extraction of Ada/SPARK Programs.* Porto, Portugal.

Oracle. (17 de 05 de 2019). *Java*. Obtenido de https://www.java.com/es/

Roby, C. (29 de 05 de 2019). *ASIS Homepage*. Obtenido de https://www.sigada.org/WG/asiswg/

The Apache Xml Project. (17 de 06 de 2019). *Xalan home page*. Obtenido de https://xml.apache.org/xalan-j/

Uppsala Universitet. (04 de 06 de 2019). *Department of Information Technology*. Obtenido de https://www.it.uu.se/

W3C . (30 de 05 de 2019). *Extensible Markup Language (XML) 1.0*. Obtenido de https://www.xml.com/axml/axml.html