Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Mejorando un sistema de soporte para la calidad de software

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Adrián Belda Romany

Tutor: Alicia Villanueva García

2015/2016

Resumen

Este trabajo se enmarca dentro de un proyecto que tiene como objetivo el análisis y verificación de propiedades de sistemas concurrentes. De forma más precisa, en este TFG se desarrolla el módulo de la aplicación dedicada a la comprobación de satisfacibilidad de una fórmula LTL que se construye en los pasos anteriores del análisis. Dado que se parte de la salida de las fases anteriores de la aplicación TADi, con una sintaxis y estructura ya decididas se ha desarrollado un parser para satisfacer las necesidades del sistema. Así pues, en este trabajo se presenta el desarrollo de la aplicación basada en el algoritmo de validación de fórmulas LTL.

Palabras clave: LTL, java, aplicación, parser.

Abstract

This work is part of a project that aims at analyzing and verifying properties of concurrent systems. More precisely, in this final degree application module dedicated to checking satisfiability of LTL formula that builds on the previous steps of the analysis is developed. Given that part of the output of the previous phases of the Tadi application with syntax and structure it has already decided parser developed to meet the needs of the system. So, in this work the development of the algorithm based on LTL formulas validation application is presented.

Keywords: LTL, java, application, parser.

Índice general

1. Introducción 8
   1.1. Motivación 8
   1.2. Objetivos 9
   1.3. Estructura de la memoria 9
2. La lógica LTL 10
   2.1. Breve introducción 10
   2.2. Parser 11
      2.2.1. Gramáticas aceptadas 11
3. Algoritmo 14
   3.1. Reglas 14
   3.2. Inconsistencias 15
   3.3. Definición del Algoritmo 15
4. Implementación 18
   4.1. Paquete computation 18
      4.1.1. Main 19
      4.1.2. Constants 19
      4.1.3. GenericMethods 21
   4.2. Paquete computation.algorithm 22
      4.2.1. Algorithm 22
      4.2.2. Rules 23
   4.3. Paquete computation.algorithm.dataStructure 24
      4.3.1. BT 24
      4.3.2. NodeBT 26
   4.4. Paquete computation.parser 28
      4.4.1. Parser 28
5. Ejecución de la aplicación 30
6. Conclusiones 35
   6.1. Consideraciones finales 35
   6.2. Trabajos futuros 35

7. Bibliografía 36
8. Anexos 37
8.1. Anexo A 37
8.2. Anexo B 37
8.3. Anexo C 38
8.4. Anexo D 39

Índice de figuras

Figura 2.1 Ejemplo operadores y modalidades. 11
Figura 3.1 Reglas α-fórmula. 14
Figura 3.2 Reglas β-fórmulas 14
Figura 4.1 Estructura de la aplicación. 18
Figura 5.1 Ejemplo ejecución 1. 30
Figura 5.2 Ejemplo ejecución 2. 30
Figura 5.3 Ejemplo ejecución 3. 32
Figura 5.4 Ejemplo ejecución 4. 34
Figura 8.1 Captura de pantalla de la ejecución 1. 37
Figura 8.2 Captura de pantalla de la ejecución 2. 37
Figura 8.3 Captura de pantalla de la ejecución 3. 38
Figura 8.4 Captura de pantalla de la ejecución 4 39

1. Introducción

En este primer capítulo se van a explicar cuáles han sido las motivaciones para la realización del proyecto así como la utilidad y necesidad de su desarrollo y los objetivos generales que éste debe cumplir. Por último se expondrá la estructura de la memoria indicando un pequeño resumen de cada uno de ellos.

1.1. Motivación

Hoy en día el desarrollo de métodos automáticos es esencial. Gracias a esto, trabajos que ahora están automatizados ahorran, por un lado, horas y horas de trabajo manuales, es decir, que proyectos que antes podían tardar 10 años por la gran complejidad a la hora de comprobar ciertas pautas de forma manual, ahora pueden realizarse en cuestión de meses o un par de años. Por otro lado, las grandes empresas también dan uso de este sistema de automatización, ya sean grandes o pequeñas empresas, consiguiendo ahorrar en personal e invirtiéndolo en otros aspectos empresariales.

En el campo de ingeniería del software, el poder automatizar un proceso es crucial. Sin embargo, no solo cuenta el hacer que las tareas se ejecuten solas, sino que también tiene que ser de forma rápida, es decir, de forma eficiente.

Una de las técnicas más conocidas para la comprobación formal es el modelo de verificación (o Model checking). Es un método automático de verificación de un sistema formal, el cual debe satisfacer una especificación formal descrita mediante una fórmula escrita en una variedad de lógica temporal. Como alternativa al model checking, en [CTV2014] se presentó un método para el análisis de propiedades LTL que cuya principal ventaja con respecto al model checking es que no necesita construir el espacio de búsqueda del sistema durante el análisis.

El principal coste computacional radica en la comprobación de satisfacibilidad de una fórmula LTL que representa el sistema y la propiedad (para más detalles ver [CTV2014]. Por lo tanto, es crucial disponer de un método eficiente que haga dicha comprobación, como el presenciado en [CTV2014, WLPE2013].

En este trabajo de fin de grado se presenta un procedimiento para verificar la validez de estas fórmulas. La lógica lineal temporal utilizada en este trabajo es una adaptación de la lógica proposicional LTL, ya que se ha enriquecido la sintaxis básica. El algoritmo de satisfacibilidad fue presentado formalmente en [WLPE2013].

1.2. Objetivos

El trabajo tiene como objetivo principal desarrollar el último paso del proyecto TADi, ya que actualmente cuenta con una implementación para la comprobación de la fórmula, pero está incompleta.

La aplicación debe implementar las siguientes funcionalidades.

1. Debe de poder realizar un parseado correcto de la fórmula que se le pasa para traducirla a una nueva que el algoritmo pueda interpretar para su resolución de forma automática.

2. Debe comprobar, de forma automática, la satisfacibilidad final para la fórmula, ya sea un estado con una fórmula satisfacible o un estado de no satisfacible.

Dado que es una aplicación para ser integrada en otra de mayor envergadura, no contará con una interfaz de usuario. Las pruebas necesarias para comprobar su funcionalidad por separado se podrán hacer a través de la consola del sistema operativo correspondiente.

1.3. Estructura de la memoria

El presente trabajo se estructura en cinco capítulos. A continuación se expondrá un breve resumen de cada uno de ellos.

- **Capítulo 1**. Se expone la motivación por la cual se ha llevado a cabo la aplicación, así como la funcionalidad y objetivos que debe abarcar.
- **Capítulo 2**. Se explica la sintaxis de las fórmulas aceptada por el parser, así como los diferentes operadores soportados por éste.
- **Capítulo 3**. En este capítulo se explica el algoritmo para automatizar la creación de tableaux LTL, tanto las reglas utilizadas por éste como la construcción del tableaux. Además, se describe con detalle la implementación realizada. **Capítulo 4**. En este capítulo se presenta un ejemplo de la ejecución y de los resultados dados, además de los tiempos de las diferentes ejecuciones.
- **Capítulo 5.** El último capítulo recoge las conclusiones obtenidas a partir del desarrollo del trabajo; analizando si los objetivos planteados han sido completados y la utilidad de la aplicación. Además, se comentarán las posibles mejoras y trabajos que puedan derivar del mismo.

2. La lógica LTL

La estructura en la que se basa la lógica temporal lineal (LTL) es un conjunto totalmente ordenado. Hace uso de una noción de tiempo discreto, con un punto inicial que no tiene predecesores e infinito. Está en una noción apropiada para razonar sobre paradigmas en general, ya que la ejecución de un programa es discreta, es decir, paso de estado a estado. Los valores realmente verdaderos de las fórmulas no se interpretan sobre un estado único, sino so sobre la secuencia de estados.

2.1. Breve introducción

Ahora se van a explicar los operadores básicos de la lógica temporal lineal proposicional y, más adelante, se explicará la gramática aceptada por nuestra aplicación. Así pues, los operadores básicos son los siguientes:

- <>ϕ Se interpreta como *en algún instante en el futuro ϕ es cierta*. En otros textos se puede encontrar este mismo operador escrito como Fϕ.
- [ ] ϕ Se lee como que *en todos los instantes futuros ϕ es cierta*. En algunos textos puede aparecer como Gϕ.
- ( ) ϕ Interpretado como *en el siguiente instante de tiempo ϕ es cierta*. Algunas veces aparece escrito como Xϕ.
- ϕ Uψ Leído como *es cierta ϕ hasta el instante futuro donde ψ es cierta*.

A continuación se presenta una figura en la que se puede observar la representación del comportamiento de los distintos operadores y modalidades. La traza *s* está representada como una secuencia de puntos y los puntos negros son el momento en el que se satisface p.

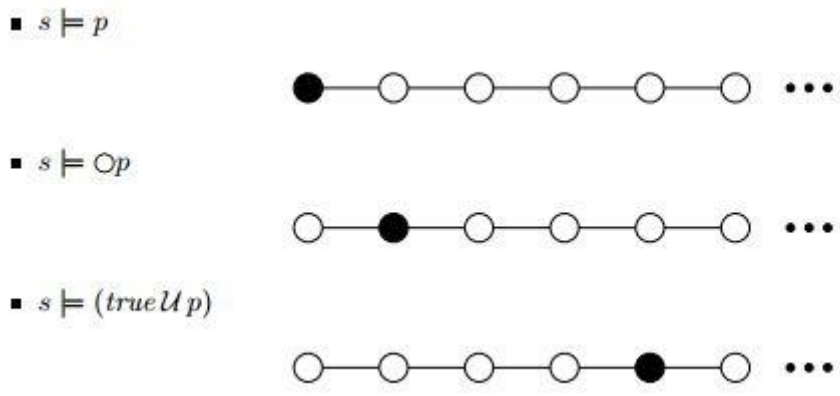

Figura 2.1 Ejemplo operadores y modalidades.

2.2. Parser

En este apartado se explica el parser que se ha desarrollado para la aplicación, teniendo en cuenta que ésta va a ser integrada en TADi.

2.2.1. Gramáticas aceptadas

A continuación, antes de empezar a explicar la gramática aceptada, se muestra una lista de abreviaturas, ya que algunos símbolos no terminantes son demasiado largos y dificultan a la hora de leer la estructura de la gramática.

- MODIFICADOR = MOD
- OPERADOR = OP
- IMPLICACION = IMP
- SENTENCIA = SNT
- NEGACION = NG

La gramática aceptada por el parser en notación BNF es la siguiente:

FORMULA ::= [*NG*] [*MOD*] *SNT* [*IMP SNT*]

MOD ::= <> | [ ] | *SIGUIENTE*

SIGUIENTE ::= NEXT | X | ( )

OP ::= AND | OR | U

IMP ::= - >

SNT ::= [*NG*] *VAR* | [*NG*] *SNT OP SNT*

VAR ::= cualquier variable (p, q, w...) | *CIERTO* | *FALSO*

CIERTO ::= TRUE | true | TT | tt

FALSO ::= FALSE | false | FF | ff

NG ::= NOT | not

Para garantizar una ejecución correcta del orden, se deben utilizar siempre paréntesis, así pues, para verlo en un ejemplo, dada la fórmula "p OR q OR w AND k", tendríamos estas variantes de ejecución:

- ((p OR q) OR w) AND k
- (p OR q) OR (w AND k)
- p OR (q OR (w AND k))

Recordar siempre el uso de los paréntesis, ya que el parser no aplica ninguna precedencia por defecto. Notese que las fases anteriores, TADi genera una fórmula bien parentizada, así que esto no supone ninguna limitación.

A continuación se van a presentar algunos ejemplos de la gramática aceptada, para reflejar de forma más clara lo expuesto arriba y despejar algunas dudas que puedan surgir. No se trata de una lista exhaustiva de fórmulas.

Fórmulas de la lógica proposicional:

- NOT p
- NOT (p AND q)
- (NOT p) -> (NOT q)
- [] (NOT p)
- p OR q
- p AND q
- (p AND q) OR (q AND w)
- (p OR q) AND (q OR w)

Fórmulas de la lógica temporal LTL:

- (() p) OR (() q)
- ((() p) U (NOT q))
- [] ((NOT p) U (<> q))

En nuestro contexto, el operador de implicación "->" sólo aparecerá una vez en la fórmula generada automáticamente por TADi, así que solo se tratará una vez.

- (NOT p) -> q
- q -> (NOT p)
- (NOT p) OR (NOT q) -> (NOT w)
- (([] (NOT p)) OR (NOT p)) -> ((NOT p) OR (<> p))

3. Algoritmo

Los algoritmos clásicos de tableaux se basan en la construcción sistemática de un grafo que se utiliza para comprobar si una fórmula se satisface correctamente o no. En [GHLN2008] se presentó un algoritmo que no necesitaba construir el grafo para luego analizarlo, sino que necesitaba construir un árbol únicamente.

La construcción del árbol se corresponde con un tableaux y se realiza a través de un procedimiento por el cual se definen ciertas reglas, las cuales serán utilizadas para construir el árbol, cuyos nodos estarán etiquetados con un conjunto de fórmulas. Si todas las ramas del árbol son *closed* (cerrado) al finalizar la ejecución del algoritmo, entonces la fórmula no tendrá ningún modelo. Sin embargo, si alguna de las ramas del árbol resulta estar *open* (abierta), eso significa que se puede obtener un modelo que satisface la fórmula por cada rama abierta.

3.1. Reglas

A cada nodo *n* se le aplica una regla dependiendo de su conjunto de fórmulas *L(n)*. La regla aplicada al nodo solamente se aplica a una fórmula ϕ del conjunto de fórmulas *L(n)*. Se diferencian dos tipos de reglas: α-fórmulas y β-fórmulas. Las alpha son conjunciones, pues mantienen un único nodo después de aplicar la regla. Por otro lado, las beta son disyunciones, ya que después de aplicar una regla de ese conjunto tendremos dos nuevos nodos diferentes. La siguiente figura representa las reglas para las α- y β-fórmulas.

	α	A(α)
R1	$\neg \neg \phi$	$\{ \phi \}$
R2	$\phi_1 \wedge \phi_2$	$\{ \phi_1, \phi_2 \}$

Figura 3.1 Reglas α-fórmula.

	β	$B_1(\beta)$	$B_2(\beta)$
R3	$\neg (\phi_1 \wedge \phi_2)$	$\{ \neg \phi_1 \}$	$\{ \neg \phi_2 \}$
R4	$\neg (\phi_1 \cup \phi_2)$	$\{ \neg \phi_1, \neg \phi_2 \}$	$\{ \phi_1, \neg \phi_2, \neg O(\phi_1 \cup \phi_2) \}$
R5	$\phi_1 \cup \phi_2$	$\{ \phi_2 \}$	$\{ \phi_1, \neg \phi_2, O(\phi_1 \cup \phi_2) \}$
R6	$\phi_1 \cup \phi_2$	$\{ \phi_2 \}$	$\{ \phi_1, \neg \phi_2, O(\Gamma \wedge \phi_1) \cup \phi_2) \}$
R7	$\phi_1 \vee \phi_2$	$\{ \phi_1 \}$	$\{ \phi_2 \}$

Figura 3.2 Reglas β-fórmulas

Cada fila de las tablas de las Figura 1 y Figura 2 representa una regla. Cuando se aplica una regla α a un nodo, la primera fórmula del conjunto *L(n)* que coincide con un patrón de la columna α se remplaza en el nodo hijo por el nuevo valor de la columna A(α). En cuanto a las reglas de β, una vez se aplica a la primera fórmula del conjunto, ésta crea dos hijos en vez de uno, el nodo izquierdo con el valor correspondiente de $B_1(\beta)$ y el nodo derecho con el valor que le corresponda de la columna $B_2(\beta)$.

En la regla R6 se encuentra una particularidad. Ésta contiene el símbolo Γ, que representa el contexto de la fórmula. Este es el mecanismo con el que se pueden encontrar bucles infinitos y, pudiendo marcar ramas como open, sin necesidad de explorarlas exhaustivamente (lo que supondría una ejecución infinita).

3.2. Inconsistencias

Antes de poder implementar el algoritmo hay que determinar las condiciones en las que los nodos son inconsistentes. Un nodo del árbol es inconsistente si el conjunto *L(n)* contiene:

- Una pareja de fórmulas contradictorias: ϕ y ¬ ϕ
- La fórmula *false*

No es posible aplicar una regla sobre un nodo inconsistente. Si una rama del árbol contiene un nodo inconsistente, esta rama será *closed*, en cualquier otro caso, la rama será *open*.

3.3. Definición del Algoritmo

Para empezar, hay que decir que el algoritmo construye un tableau de LTL automáticamente a partir de un conjunto de fórmulas.

De forma breve se puede decir que esta construcción consiste en seleccionar una rama del árbol que todavía no esté expandida al máximo y, aplicando las reglas α y β, determinar la continuidad de cada nodo de esa rama. Cuando a un nodo no se le puede aplicar ninguna de esas reglas, se aplica, si existe alguno en el conjunto de fórmulas, el operador *next*.
Una vez no puede aplicarse ninguna regla y no hay ningún operador *next*, si la hoja de esa rama no contiene ninguna inconsistencia, la rama se marca como *open*, en caso contrario se marca como *closed*.

De forma más detallada, el algoritmo es más complejo que lo dicho anteriormente. Dado un conjunto finito de fórmulas, y siendo este primer conjunto el

nodo padre del árbol, se recorre cada nodo hoja *h*, formado por el conjunto de fórmulas *L(h)*, aplicando los siguientes pasos de forma ordenada:

1. Se comprueban las fórmulas del conjunto *L(h)* del nodo *h* para verificar si hay o no una inconsistencia. Si la hay, se marca el nodo *h* como *closed*.

2. Si el *L(h)* es un conjunto de fórmulas de proposiciones atómicas, es decir, está formado por fórmulas básicas a las cuales no se les puede aplicar ninguna regla, se marca el nodo *h* como *open*.

3. Si *L(h) = L(h')* siendo *h'* antecesor a *h*, esto es, si el conjunto de fórmulas del nodo coincide con el conjunto de fórmulas de un nodo antecesor suyo, entonces se busca al nodo más antiguo *h''* (más cercano al a raíz dentro de su rama) para el cual el conjunto de fórmulas es el mismo.
   Se comprueba si todas las fórmulas del conjunto que están formadas por un UNTIL o un EVENTUALLY han estado distinguidas tanto en el nodo *h''* como en el nodo *h*, además de los nodos intermedios que haya entre estos dos. Si esta comprobación es correcta, significa que todas las fórmulas que tienen un UNTIL y/o EVENTUALLY han sido procesadas y por lo tanto el nodo *h* se marca como *open*.

4. Si no se cumple ninguna condición de los pasos anteriores, entonces se elige una fórmula $\phi \in L(h)$, pero asegurándose de que ϕ no sea una fórmula NEXT.

   - Si ϕ es una fórmula α, se crea un nuevo nodo *h'* como hijo de *h*, en el cual el conjunto de fórmulas se transforma según se haya aplicado la regla R1 o R2 de la tabla de la Figura 1.

   - Si ϕ es una fórmula β, se crean dos nuevos nodos *h'* y *h''* como hijos del nodo *h*, cuyos conjuntos de fórmulas se definen siguiendo las reglas R3-R7 de la Figura 2. Si además la regla es una eventualidad (se va a aplicar a un UNTIL o EVENTUALLY), hay tres posibles casos:

     a. Si β es la fórmula distinguida en *L(h)*, entonces se aplica la regla R6 a β. En el nodo izquierdo, *h'*, la fórmula distinguida será igual a la del padre, *h*, y en el nodo derecho, *h''*, se distingue la fórmula que está dentro del NEXT en $B_2(\beta)$.

     b. Si β no es la fórmula distinguida, pero hay una fórmula distinguida, entonces se aplica la regla R5 a β. Los nuevos nodos heredan de *h* la fórmula distinguida.

     c. Si no se cumple ninguno de los dos casos anteriores, entonces β pasa a ser la nueva fórmula distinguida en el nodo *h* y se

aplica la regla R6. Así pues, *h'* tendrá la misma fórmula distinguida que *h* y para *h''* se distingue la fórmula que está dentro del NEXT en $B_2(\beta)$.

- Si en el conjunto *L(h)* solo quedan fórmulas NEXT o proposiciones atómicas, entonces simplemente se aplica el operador NEXT. Esto consiste en crear un nodo *h'* hijo de *h*, el cual contendrá en su conjunto *L(h')* las fórmulas que hay dentro del operador u operadores NEXT del nodo padre. Además heredará la fórmula distinguida de *h*.

Una vez todos los nodos hoja están marcados como *open* o *closed*, la construcción de la tableaux finaliza y con ello la ejecución del algoritmo.

4. Implementación

En este apartado se va a describir la implementación realizada para llevar a cabo el desarrollo del algoritmo. Se ha utilizado el lenguaje de programación Java y se ha dividido la estructura en 4 paquetes.

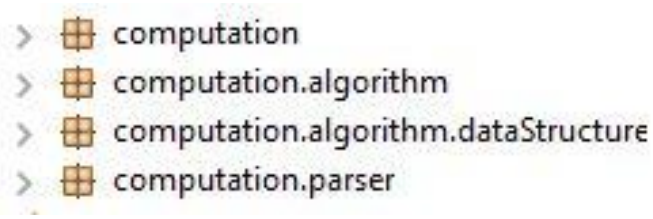

Figura 4.1 Estructura de la aplicación.

A continuación se va comentar los contenidos de cada paquete. La documentación se ha realizado en inglés, ya que se ha buscado la internacionalización del proyecto y facilita la continuidad del trabajo.

4.1. Paquete computation

En este paquete está implementada la clase main, que dará paso a la ejecución de toda la aplicación. Además, todas aquellas clases que serán utilizadas desde las otras clases, es decir, aquellas clases que serán accedidas para consultar métodos o constantes, también están definidas en este paquete. Las clases integradas en este paquete son las siguientes:

- Main.java
- Constants.java
- GenericMethods.java

4.1.1. Main

public class **Main**

extends Object

Desde este método principal se llamará al parser y, posteriormente, al algoritmo.

Method Summary	
static void	main(String[] args) Method which will launch the execution of the application.

4.1.2. Constants

public class **Constants**

extends Object

En esta clase están implementadas todas las constantes utilizadas por el programa. Con esto conseguimos que, en caso de tener que modificar el valor de un operador, solo sería necesario cambiarlo en esta clase.

Method Summary	
static List<String>	getListBasicallyOp() Returns the value of the listBasicallyOp constant
static List<String>	getListMod() Returns the value of the listMod constant
static List<String>	getListNext() Returns the value of the OP_NOT constant
static List<String>	getListNotBasicallyOp() Returns the value of the listNotBasicallyOp constant
static List<String>	getListNotConstraintFormulas() Returns the value of the listNotConstraintFormulas constant
static List<String>	getListSV() Returns the value of the listSV constant
static List<String>	getListSVFalse() Returns the value of the listSVFalse constant

static List<String>	**getListSVTrue**() Returns the value of the listSVTrue constant
static String	**getMDEventually**() Returns the value of the MD_EVENTUALLY constant
static String	**getMDGenerally**() Returns the value of the MD_GENERALLY constant
static String	**getMDImplies**() Returns the value of the MD_IMPLIES constant
static String	**getOPAND**() Returns the value of the OP_AND constant
static String	**getOPAnd**() Returns the value of the OP_and constant
static String	**getOPNext**() Returns the value of the OP_NEXT constant
static String	**getOPNext1**() Returns the value of the OP_NEXT1 constant
static String	**getOPNext2**() Returns the value of the OP_NEXT2 constant
static String	**getOPNOT**() Returns the value of the OP_NOT constant
static String	**getOPNot**() Returns the value of the OP_not constant
static String	**getOPOR**() Returns the value of the OP_OR constant
static String	**getOPOr**() Returns the value of the OP_or constant
static String	**getOPPower**() Returns the value of the OP_POWER constant
static String	**getOPUntil**() Returns the value of the OP_UNTIL constant
static String	**getSVFALSE**() Returns the value of the SV_FALSE constant
static String	**getSVFalse**() Returns the value of the SV_false constant

static String	**getSVFF**() Returns the value of the SV_FF constant
static String	**getSVff**() Returns the value of the SV_ff constant
static String	**getSVTRUE**() Returns the value of the SV_TRUE constant
static String	**getSVTrue**() Returns the value of the SV_true constant
static String	**getSVTT**() Returns the value of the SV_TT constant
static String	**getSVtt**() Returns the value of the SV_tt constant

4.1.3. GenericMethods

```
public class GenericMethods

extends Object

Clase con métodos que serán usados a lo largo de toda la aplicación para
diferentes comprobaciones.
```

Method Summary	
static String	**listToString**(List<String> list) Given a list of strings builds a single String from this.
static boolean	**sentenceContainedInParenthesis**(String sentence) Checks if a sentence is grouped within a parenthesis or not.
static List<String>	**separateEquationSpacesAndParenthesis**(String equToGet) Divide the equation by spaces and parentheses to treat it.
static String	**simplifyEquationBasicArit**(String equation) Method to simplify the basic operations with "true" and "false".

4.2. Paquete computation.algorithm

Este es el paquete más importante, ya que en él están las clases principales que darán vida al algoritmo. Las clases contenidas en este paquete son:

- Algorithm.java
- Rules.java

Como se puede deducir, las reglas y el algoritmo en si están separados en dos clases diferentes. La clase Algorithm.java tiene implementado todo aquello descrito en la sección 3.3, mientras que la clase Rules.java tiene implementadas las reglas definidas en las figuras 3.1 y 3.2 del apartado 3.1, cada regla corresponde a un método. Además de la implementación de las reglas, también se han desarrollado ciertos métodos para comprobar si hay que utilizar una regla u otra, a pesar de que estos métodos de comprobación son llamados desde la clase Algorithm.java.

Esta implementación está pensada para crear una estructura clara de lo que es cada cosa y buscando una mayor facilidad a la hora de tener que actualizar la aplicación en el futuro. Si se quisiera modificar por cualquier motivo, la estructura proporciona una clara idea de dónde debería añadirse esta nueva regla.

4.2.1. Algorithm

```
public class Algorithm

extends Object

Clase en la que se desarrolla el algoritmo para verificar la validez de las
fórmulas.
```

Method Summary	
static boolean	**checkAntecesorNodes**(NodeBT node) This method checks if the set of formulas of the actual node is equal to some other set of formulas of their past nodes.
static BT<String>	**executeAlgorithm**() Private method who is called by the public method to execute the algorithm.
static BT<String>	**executeAlgorithm**(String equation) Public method which constructs the Binary Tree of the equation received and starts executing the algorithm.
static boolean	**formulaStartWithNextOrNot**(String formula) This method checks if it's a NEXT formula or not.

static boolean	**isConstraintFormulasOrNot**(NodeBT node) This method checks if all formulas of the node that receives are constraint formulas or not.
static boolean	**isInconsistentNodeOrNot**(NodeBT node) This method checks if the node that receives is inconsistent or not.

4.2.2. Rules

```
public class Rules

extends Object

En esta clase están definidas las reglas por las que se rige el algoritmo.
```

Method Summary	
static Boolean	**isAlphaFormulaOrNot**(String equation) This method checks if it's an Alpha formula.
static Boolean	**isBetaFormulaOrNot**(String equation) This method checks if is a Beta formula.
static Boolean	**isRule1OrNot**(String equation) This method checks if it's Rule 1.
static Boolean	**isRule2OrNot**(String equation) This method checks if it's Rule 2.
static Boolean	**isRule3OrNot**(String equation) This method checks if it's Rule 3.
static Boolean	**isRule4OrNot**(String equation) This method checks if it's Rule 4.
static Boolean	**isRule5Or6OrNot**(String equation) This method checks if it's Rule 5 or Rule 6.
static Boolean	**isRule7OrNot**(String equation) This method checks if it's Rule 7.
static String	**rule1**(String equation) This method applied the Rule 1 to the equation received.
static List<String>	**rule2**(String equation) This method applied the Rule 2 to the equation received.
static List<List<String>>	**rule3**(String equation) This method applied the Rule 3 to the equation received.

static List<List<String>>	**rule4**(String equation) This method applied the Rule 4 to the equation received.
static List<List<String>>	**rule5**(String equation) This method applied the Rule 5 to the equation received.
static List<List<String>>	**rule6**(String equation, List<String> gamma) This method applied the Rule 6 to the equation received.
static List<List<String>>	**rule7**(String equation) This method applied the Rule 7 to the equation received.

4.3. Paquete computation.algorithm.dataStructure

En este paquete está implementada la estructura en la cual se van a guardas los datos y pasos del algoritmo. Está dividido en las siguientes clases:

- BT.java
- NodeBT.java

BT hace referencia a BinaryTree. Se utiliza esta estructura porque, como ya se ha explicado anteriormente en el algoritmo, cada nodo puede tener uno o dos hijos. La clase BT.java tiene implementada la estructura de los Binary Tree con los métodos más globales. Así pues la clase NodeBT.java implementa la estructura interna de los nodos.

4.3.1. BT

```
public class BT<E extends Comparable<E>>

extends Object

Esta clase implementa los métodos necesarios para tener una estructura de
árbol.
```

Constructor Summary
BT() Constructor of an empty BT.
BT(List<String> listaFormulas) Constructor of a BT with formulas.

Method Summary	
String	**getAlgorithmResult**() Returns the value of the algorithmResult attribute.
boolean	**isEmpty**() Returns if the BT is empty.
NodeBT	**recover**() Returns the next node to be used by the algorithm.
NodeBT	**recover**(NodeBT nodo) Called by the public method recover.
String	**resultToString**(NodeBT node) Returns a String representation of the branch.
void	**setAlgorithmResult**(String algorithmResult) Sets the value of the algorithmResult attribute.
void	**setSize**(int newSize) Sets the value of the size attribute.
int	**size**() Returns the size of the current binary tree.
String	**subtreeToString**(NodeBT node, int numNode) Called by the public method toString.
String	**toString**() Returns a String representation of the contents of the set.

4.3.2. NodeBT

```
public class NodeBT

extends Object
```

Clase que define la estructura interna de los nodos. Cada nodo hace referencia a un estado por el que pasa la fórmula.

Constructor Summary	
NodeBT(List<String> formulaList) Constructor of a node without child.	
NodeBT(List<String> formulaList, NodeBT right, NodeBT left, NodeBT parent, String distinguish, int mark, String appliedRule) Constructor of a node with all their attributes.	
NodeBT(List<String> formulaList, NodeBT parent, String distinguish) Constructor of a node that have a parent and a distinguish sentence.	
Method Summary	
String	getAppliedRule() Returns the value of the appliedRule attribute.
String	getDistinguish() Returns the value of the distinguish attribute.
List<String>	getFormulaList() Returns the value of the formulaList attribute.
NodeBT	getLeft() Returns the value of the left attribute.
int	getMark() Returns the value of the mark attribute.
NodeBT	getParent() Returns the value of the parent attribute.
NodeBT	getRight() Returns the value of the right attribute.
String	resultToString() Returns the attributes of the node in a String.
void	setAppliedRule(String appliedRule) Sets the value of the appliedRule attribute.

void	**setDistinguish**(String distinguish) Sets the value of the distinguish attribute.
void	**setFormulaList**(List<String> formulaList) Sets the value of the formulaList attribute.
void	**setLeft**(NodeBT left) Sets the value of the left attribute.
void	**setMark**(int mark) Sets the value of the mark attribute.
void	**setParent**(NodeBT parent) Sets the value of the parent attribute.
void	**setRight**(NodeBT right) Sets the value of the right attribute.

Field Summary	
List<String>	**formulaList** List that will contain the formulas of the node.
NodeBT	**right** Pointer to a node of type $A(\alpha)$ or to one of type $B_1(\beta)$
NodeBT	**left** Pointer to a node of type $B_2(\beta)$
NodeBT	**parent** Pointer to the parent node
String	**distinguish** Formula distinguished in the node
int	**mark** Indicates whether the node must be processed (0), is open (1), closed (-1) or is not a leaf node (2)
String	**appliedRule** It indicates the rule that has been used in this node

4.4. Paquete computation.parser

En este paquete se ha desarrollado el parser. Así pues, la clase que se encuentra en este paquete será la siguiente:

- Parser.java

No hay más clases implementadas, ya que con esta y las dos generales del paquete computation es suficiente para poder llevar a cabo su función.

4.4.1. Parser

```
public class Parser

extends Object
```

Clase que simplifica y estructura la fórmula recibida para pasarla al algoritmo.

Method Summary	
String	**getModifier**(HashMap<String> hashEquation) This method returns the modifier of the equation.
String	**getOperator**(HashMap<String> hashEquation, int numberOperator) This method returns the requested operator inside of the equation.
String	**getSentence**(HashMap<String> hashEquation, int numberSentence) This method returns the requested sentence inside of the equation.
int	**getTotalSentences**(HashMap<String> hashEquation) This method returns the total sentences inside of the equation.
String	**joinEquation**(HashMap<String> hashEquation) This method constructs the equation (in String format) from the data contained in the equation (HashMap format) that is received as a parameter.
String	**readEquation**(String equParam) Public method that calls the private method with the necessary data to process the equation to simplify it and to construct the correct structure.
String	**readEquation**(String equParam, boolean firstInput, int lastSentence) Private method with the necessary data to process the equation to simplify it and to construct the correct structure.
void	**separateEquation**(String currentEquation) This method separates the equation in modifier (if any), sentences and operators.

void	**simplifyEquation**(int indexOfHashEquationToUpdate) This method extracts the equation of the global list thanks to the index is received as a parameter.
String	**simplifyEquationByNot**(String equation) Method to simplify NOT operators of the final equation.
void	**valueUpdate**(int indexOfHashEquationToUpdate, int indexOfSentenceToReplace, int lastSentence) This method update the value of the sentence of the hash of a position of the list "equation" (global parameter of the class)

5. Ejecución de la aplicación

En este apartado se van a describir varios ejemplos de ejecución de la aplicación. Además, se podrá comprobar que los tiempos de ejecución son bajos y que no aportarán apenas un coste adicional a la hora de que sea implantado en el proyecto TADi.

a) En este primer ejemplo se ejecutó la sentencia “p AND (NOT p)”.

```
p AND (NOT p)

The formula, which is an input for the algorithm:
p AND (NOT p)

Algoritmo:
The formula is non satisfiable
Runtime in milliseconds: 10
```

Figura 5.1 Ejemplo ejecución 1.

Es una sentencia bastante básica y, como se puede apreciar, no tiene ningún modelo asociado, ya que es una fórmula no satisfacible para el algoritmo.

Por otro lado, se puede apreciar que apenas tarda 10 milisegundos en ejecutar tanto el parseado como el propio algoritmo.

En el anexo A se puede observar una captura de pantalla de la ejecución.

b) En este segundo ejemplo se ejecutó la sentencia “NOT (([] p) -> (<> p))”.

```
NOT (([] p) -> (<> p))

The formula, which is an input for the algorithm:
(NOT (tt U (NOT p))) AND (NOT (tt U p))

Algoritmo:
The formula is non satisfiable
Runtime in milliseconds: 16
```

Figura 5.2 Ejemplo ejecución 2.

Es una sentencia algo más compleja y, como la del anterior ejemplo, no tiene ningún modelo asociado.

Se puede apreciar que apenas tarda 16 milisegundos en ejecutar tanto el parseado como el propio algoritmo.

En el anexo B se puede observar una captura de pantalla de la ejecución.

c) En este tercer ejemplo se ejecutó la sentencia “p -> (<> p)”.

Esta sentencia es bastante similar a la anterior, pero esta vez si tiene un modelo asociado, es decir, es satisfacible.

En este caso, se puede comprobar que el tiempo de ejecución es de 17 milisegundos. Es un tiempo mayor que en los dos anteriores casos, pero sigue siendo un tiempo bastante pequeño para la futura integración con el proyecto TADi.

En el anexo C se puede observar una captura de pantalla de la ejecución.

```
p -> (<> p)

The formula, which is an input for the algorithm:
NOT (p AND (NOT (tt U p)))

Algoritmo:
The formula is satisfiable. A model/s for the formula is/are:
Branch 1:
NodeBT:
      L(l) = [NOT (p AND (NOT (tt U p)))]
      Formula distinguished = none
      Applied rule = Rule 3
NodeBT:
      L(l) = [NOT p]
      Formula distinguished = none
      Mark = open
Branch 2:
NodeBT:
      L(l) = [NOT (p AND (NOT (tt U p)))]
      Formula distinguished = none
      Applied rule = Rule 3
NodeBT:
      L(l) = [NOT (NOT (tt U p))]
      Formula distinguished = none
      Applied rule = Rule 1
NodeBT:
      L(l) = [tt U p]
      Formula distinguished = tt U p
      Applied rule = Rule 6
NodeBT:
      L(l) = [p]
      Formula distinguished = tt U p
      Mark = open
Branch 3:
NodeBT:
      L(l) = [NOT (p AND (NOT (tt U p)))]
      Formula distinguished = none
      Applied rule = Rule 3
NodeBT:
      L(l) = [NOT (NOT (tt U p))]
      Formula distinguished = none
      Applied rule = Rule 1
NodeBT:
      L(l) = [tt U p]
      Formula distinguished = tt U p
      Applied rule = Rule 6
NodeBT:
      L(l) = [tt, NOT p, NEXT (ff U p)]
      Formula distinguished = ff U p
      Applied rule = None. NEXT operator applied
NodeBT:
      L(l) = [ff U p]
      Formula distinguished = ff U p
      Applied rule = Rule 6
NodeBT:
      L(l) = [p]
      Formula distinguished = ff U p
      Mark = open

Runtime in milliseconds: 17
```

Figura 5.3 Ejemplo ejecución 3.

Respecto a la salida, puede verse la fórmula simplificada por el parser, la cual procesará el algoritmo. A continuación aparece el resultado del algoritmo, el cual devolverá si es o no satisfacible. La traza de las ramas se lee de arriba abajo, siendo el de arriba el nodo raíz y el último el hoja. En caso de que lo sea devolverá también las ramas cuyos nodos hoja están abiertos. Por cada nodo se podrán comprobar los siguientes parámetros de salida:

- Conjunto de fórmulas.
- La fórmula distinguida.
- La regla que se ha aplicado a dicho nodo (excepto si es un nodo hoja, que no tendrá). Si aparece "NEXT operator applied" significa que se ha aplicado tercer punto del cuarto paso del algoritmo, es decir, el último paso en cada iteración del algoritmo.
- El atributo *mark*, que indicará que el nodo está abierto (ya que solo se muestran los nodos abiertos). Éste solo aparecerá en los últimos, los nodos hoja.

Por último se puede observar una línea indicando el tiempo de ejecución. Esta incluye tanto el tiempo de ejecución del parser como el tiempo empleado en el algoritmo. Así pues podemos decir que no es un coste muy elevado.

d) En este último ejemplo se ejecutó la sentencia "p U (NOT w)".

Se puede comprobar que el tiempo de ejecución es de 22 milisegundos. Dado que la complejidad radica en este tipo de fórmulas, podrían dar lugar a ejecuciones infinitas si no se tratasen de forma adecuada.

En el anexo D se puede observar una captura de pantalla de la ejecución.

```
p U (NOT w)

The formula, which is an input for the algorithm:
p U (NOT w)

Algoritmo:
The formula is satisfiable. A model/s for the formula is/are:
Branch 1:
NodeBT:
      L(l) = [p U (NOT w)]
      Formula distinguished = p U (NOT w)
      Applied rule = Rule 6
NodeBT:
      L(l) = [NOT w]
      Formula distinguished = p U (NOT w)
      Mark = open
Branch 2:
NodeBT:
      L(l) = [p U (NOT w)]
      Formula distinguished = p U (NOT w)
      Applied rule = Rule 6
NodeBT:
      L(l) = [p, NOT (NOT w), NEXT ((ff AND p) U (NOT w))]
      Formula distinguished = (ff AND p) U (NOT w)
      Applied rule = Rule 1
NodeBT:
      L(l) = [p, NEXT ((ff AND p) U (NOT w)), w]
      Formula distinguished = (ff AND p) U (NOT w)
      Applied rule = None. NEXT operator applied
NodeBT:
      L(l) = [(ff AND p) U (NOT w)]
      Formula distinguished = (ff AND p) U (NOT w)
      Applied rule = Rule 6
NodeBT:
      L(l) = [NOT w]
      Formula distinguished = (ff AND p) U (NOT w)
      Mark = open

Runtime in milliseconds: 16
```

Figura 5.4 Ejemplo ejecución 4.

6. Conclusiones

En este último capítulo se plantearan las consideraciones obtenidas a partir del trabajo desarrollado para esto se realizará un repaso de los diferentes objetivos que se plantearon al inició de este proyecto y cuáles de ellos se han conseguido completar y con qué grado de acierto. Además, se explicaran las posibles ampliaciones que pueden llevarse a cabo en el proyecto con el objetivo de mejorar su funcionalidad o su aspecto visual.

6.1. Consideraciones finales

Haciendo un repaso de los objetivos que fueron planteados al inicio del proyecto y los resultados obtenidos, podemos concluir que la aplicación resuelve el problema presentado.

La aplicación es capaz de transformar correctamente, a través del parser, una fórmula presentada y, posteriormente, aplicar el algoritmo para averiguar si la fórmula dada es satisfacible o no y, en caso de que lo sea, determinar, como mínimo, un modelo para la fórmula.

Por otro lado, se ha comprobado que los tiempos de cómputo son buenos. Así pues para la futura integración con TADi, además de ser una aplicación completa, contarán con una aplicación eficaz en rendimiento.

6.2. Trabajos futuros

Como trabajo futuro a llevar a cabo a partir de este proyecto, debería de terminarle la integración a la plataforma TADi, ya que el principal objetivo de este proyecto era realizar la implementación del último paso, pero la parte de la implantación se consideró para un trabajo futuro.

Por otro lado, también podría realizarse una interfaz de usuario para hacer de esta una aplicación independiente, para todo aquel que necesite solo la aplicación del algoritmo en sus propios proyectos.

7. Bibliografía

[1] Java™ Platform (2015). Java™ Platform, Standard Edition 7 API Specification <http://docs.oracle.com/javase/7/docs/api/>

[2] [WLPE2013] Marco Comini, Laura Titolo, Alicia Villanueva (2013). Towards an Effective Decision Procedure for LTL formulas with Constraints <http://arxiv.org/pdf/1308.4171v1.pdf>

[3] Wikipedia (2015). Linear temporal logic <https://en.wikipedia.org/wiki/Linear_temporal_logic>

[4] University of Twente (2009). LTL Model Checking <https://people.irisa.fr/Axel.Legay/lec3.pdf>

[5] Daniel Fernández (2010). Usando DocFlex/Javadoc en Eclipse <https://dafero.wordpress.com/2010/09/26/docflex-javadoc-eclips/>

[6] [CTV2014] Marco Comini, Laura Titolo, Alicia Villanueva (2014). Abstract Diagnosis for tccp using a Linear Temporal Logic. TPLP 14(4-5): 787-801 (2014) <http://arxiv.org/pdf/1405.3675v1.pdf>

[7] [GHLN2008] Joxe Gaintzarain, Montserrat Hermo, Paqui Lucio, Marisa Navarro (2008) Systematic Semantic Tableaux for PLTL. Electr. Notes Theor. Comput. Sci. 206: 59-73 <http://www.sc.ehu.es/jiwlucap/prole08b.pdf>

[8] Alicia Villanueva García (2006) Apuntes de la Asignatura Herramientas Avanzadas para el Desarrollo de Software v1.0 <http://www.dsic.upv.es/docs/bib-dig/documentacion/etd-08242006-162558/ReportHAD.pdf>

8. Anexos

8.1. Anexo A

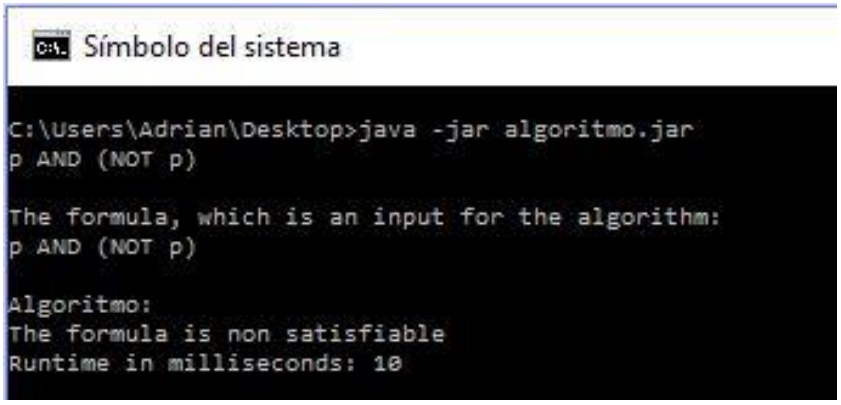

Figura 8.1 Captura de pantalla de la ejecución 1.

8.2. Anexo B

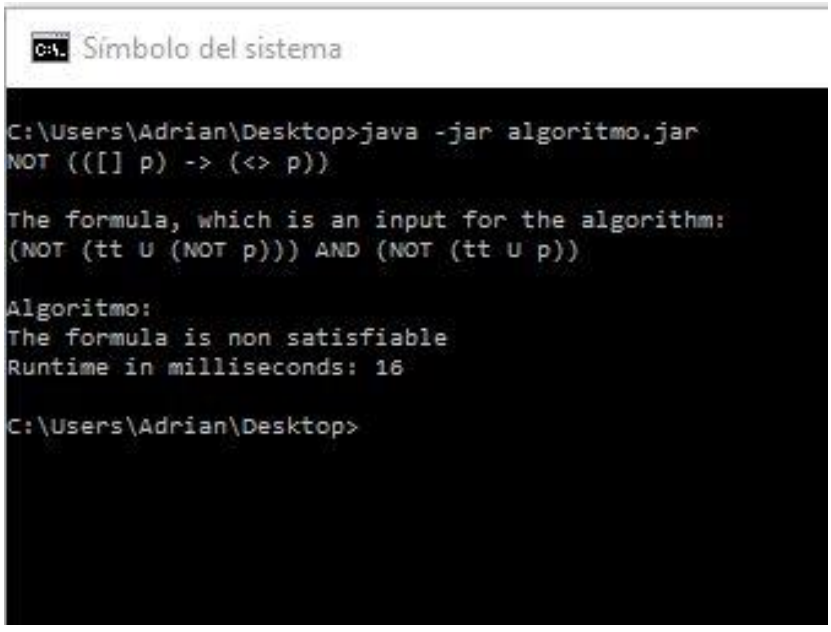

Figura 8.2 Captura de pantalla de la ejecución 2.

8.3. Anexo C

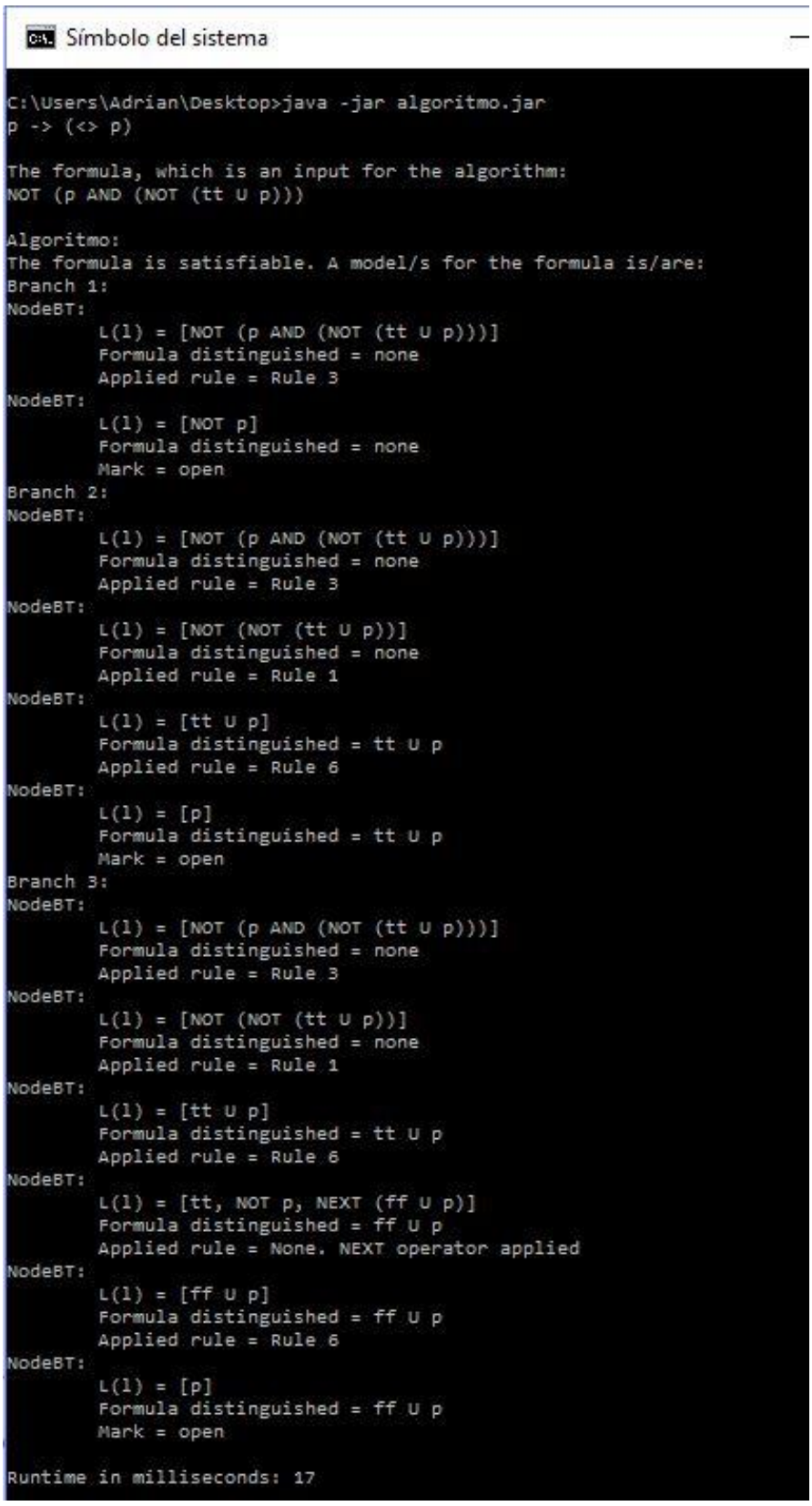

Figura 8.3 Captura de pantalla de la ejecución 3.

8.4. Anexo D

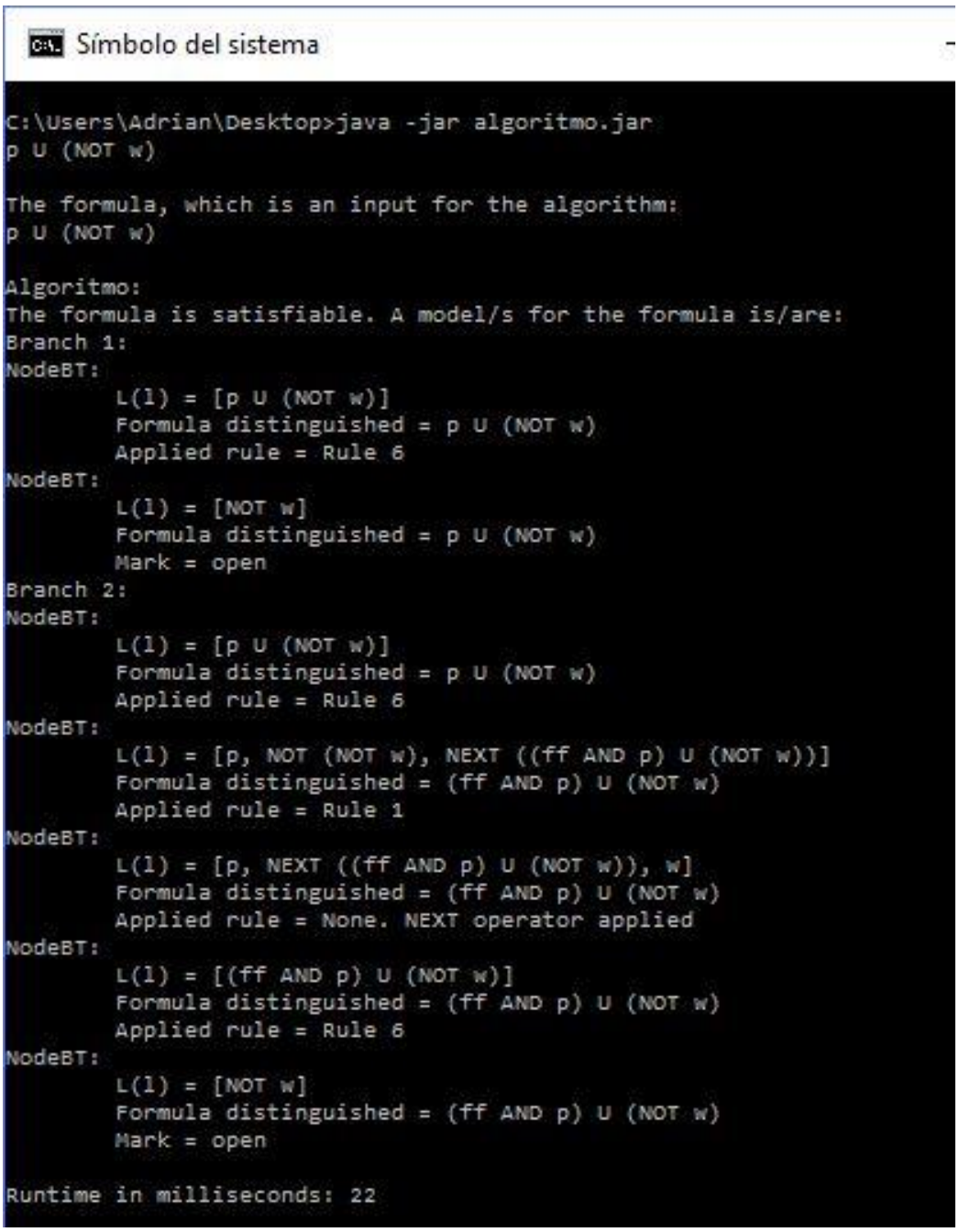

Figura 8.4 Captura de pantalla de la ejecución 4