UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de una herramienta de escritorio para depuración de programas basada en fragmentación de trazas

Trabajo fin de grado

Grado en Ingeniería Informática

Autor: Cristina Requena Casares

Tutor: María Alpuente Frasnedo
Julia Sapiña Sanchis

Curso 2015-2016

Resumen

El concepto de aplicación web está relacionado con el almacenamiento en la nube y el acceso a la información vía Internet, quedando una copia temporal dentro de nuestro dispositivo. Sin embargo, las aplicaciones web no pueden garantizar ciertos beneficios en cuanto a programas grandes se refiere.

Las aplicaciones de escritorio, también llamadas aplicaciones *desktop*, suelen estar desarrolladas acorde a un sistema operativo, pudiendo acceder más fácilmente al hardware del equipo y así tener un mejor tiempo de respuesta. Sin embargo, los lenguajes utilizados en la programación *online* suelen ser interpretados o pseudointerpretados, por lo que tienen como inconveniente un peor rendimiento. También requieren de un menor tiempo de desarrollo y, por consiguiente, menor coste, dado que las herramientas de desarrollo y depuración para la programación tradicional están más desarrolladas.

ANIMA es un depurador de programas escritos en lenguaje Maude, que implementa eficientemente la lógica de reescritura (RWL). La técnica de depuración implementada en ANIMA se basa en la fragmentación de trazas (*trace slicing*) y en técnicas de visualización, habiendo sido desarrollada como una aplicación on-line que despliega interactivamente el árbol de computación del programa que se pretende depurar.

Este trabajo de fin de grado se ha centrado en el desarrollo de la aplicación *d*ANIMA (*desktop* ANIMA), una versión de escritorio y multiplataforma de la aplicación *online* ANIMA. *d*ANIMA ha sido implementada utilizando Java como lenguaje principal de manera que pueda ejecutarse en diferentes sistemas operativos (p. ej., Windows, OS X, Linux) y al mismo tiempo beneficiarse de la eficiencia característica que las tradicionales aplicaciones de escritorio ofrecen frente a las aplicaciones *online*, limitadas en muchas ocasiones por el propio navegador.

Palabras clave: Métodos Formales en Ingeniería del Software, Lenguaje de programación, Análisis y Depuración de Programas

Resum

El concepte d'aplicació web està relacionat amb l'emmagatzemament en el núvol i l'accés a la informació via Internet, quedant una còpia temporal dins del nostre dispositiu. No obstant això, les dites aplicacions no poden garantir certs beneficis quant a programes grans es referix.

Les aplicacions d'escriptori solen estar desenvolupades d'acord amb un sistema operatiu, podent accedir mes fàcilment al maquinari de l'equip i així tindre un millor temps de resposta. No obstant això, els llenguatges utilitzats en la programació *online* solen ser interpretats o pseudointerpretats, per la qual cosa tenen com a inconvenient un pitjor rendiment. També requerixen d'un menor temps de desenvolupament i, per consegüent, menor cost, atés que les ferramentes de desenvolupament i depuració per a la programació tradicional estan mes desenvolupades.

ANIMA és un depurador de programes escrits en llenguatge Maude, que implementa eficientment la lògica de reescriptura (RWL). La tècnica de depuració implementada en ANIMA es basa en la fragmentació de traces (trace slicing) i en tècniques de visualització, havent sigut desenvolupada com una aplicació *online* que desplega interactivament l'arbre de computació del programa que es pretén depurar.

Este treball de fi de grau s'ha centrat en el desenvolupament de l'aplicació *d*ANIMA (*desktop* ANIMA), una versió d'escriptori i multiplataforma de l'aplicació *online* ANIMA. *d*ANIMA ha sigut implementada utilitzant Java com a llenguatge principal de manera que puga executar-se en diferents sistemes operatius (p. ex., Windows, OS X, Linux) i al mateix temps beneficiar-se de l'eficiència característica que les tradicionals aplicacions d'escriptori oferixen enfront de las aplicacions *online*, limitades en moltes ocasions pel propi navegador.

Paraules clau: Mètodes Formals en Enginyeria del Programari, Llenguatge de Programació, Anàlisi i Depuració de Programes

Abstract

Web applications offer involve the storage of information in the cloud and access to it through the Internet, keeping a temporary copy inside the user's computer. Unfortunately, for the case of large programs, web applications cannot guarantee some important demands.

Desktop applications are usually developed according to an operating system, being able to more easily access the hardware, and this way they offer better performance. The languages used in *online* programming are usually interpreted or pseudointerpreted. They require less development time and, consequently, less cost, since also the development hardware and tools are more sophisticated for traditional programming.

ANIMA is a debugger for programs written in the Maude language that efficiently implements Rewriting Logic (RWL). The debugging techniques implemented in ANIMA are based on trace slicing and visualization techniques, having been developed as an *online* application that interactively displays the computation tree for the considered program we want to debug.

This Final Degree Project has focused on developing the *d*ANIMA (*desktop* ANIMA) application, a desktop and cross-platform *online* version of the original application ANIMA. *d*ANIMA has been implemented using Java as the main language so that it runs on different operating systems (e.g., Windows, OS X, Linux) and at the same time benefits from the characteristic efficiency offered by traditional desktop applications in comparison to *online* applications limited in many occasions by the browser itself.

Key words: Formal Methods in Software Engineering, Programming Languages, Program Analysis and Debugging

Índice general

Índice de figuras

Índice de tablas

Agradecimientos

Quiero agradecer a mi compañero Rubén y a mis tutoras María y Julia por el tiempo que han invertido y el apoyo recibido durante la realización de este proyecto.

Glosario

API *Application Programming Interface.* Interfaz de programación de aplicaciones: es el conjunto de funciones y procedimientos que ofrece una biblioteca para ser utilizado por otro software como una capa de abstracción.

Framework Estructura conceptual y tecnológica de soporte definido que puede servir de base para la organización y desarrollo de software. Puede incluir soporte de programas, bibliotecas y un lenguaje interpretado, entre otras herramientas, y así servir de ayuda para desarrollar y unir las diferentes partes de un proyecto.

JSON *JavaScript Object Notation.* Formato ligero de intercambio de datos, completamente independiente del lenguaje de programación.

JVM *Máquina Virtual Java.* Máquina virtual de proceso nativo, es decir, ejecutable en una plataforma específica, capaz de interpretar y ejecutar instrucciones expresadas en un código binario especial (el bytecode Java), el cual es generado por el compilador del lenguaje Java.

UI *Interfaz de usuario.* Medio con el que el usuario puede comunicarse con una máquina, equipo, computadora o dispositivo, y comprende todos los puntos de contacto entre el usuario y el equipo.

Menú Es la lista o relación de programas y procedimientos que aparece en pantalla con el fin de que, usando un teclado, un dispositivo táctil, un lápiz óptico o un ratón, el operador pueda elegir qué opción desea ejecutar. Los denominados *menús desplegables* son el tipo más habitual en la actualidad: muestran un Menú Principal y, tras una sencilla operación, sus diferentes opciones.

Menú Contextual Listado de definiciones que se pueden seleccionar cuando se pulsa con el botón secundario del ratón (generalmente el derecho) sobre un determinado elemento o área de la ventana, de un programa. Dichas opciones suelen estar escogidas dentro de un contexto, es decir, seleccionadas para acceder directamente a determinadas áreas del programa.

Toolbar *Barra de herramientas.* Es una fila o columna de botones en pantalla que se utiliza para activar diversas funciones de la aplicación. Por lo general, la barra es movible, de manera que puede colocarse cerca del objeto en que se está trabajando para cambiar rápidamente modos y opciones. Además estas herramientas pueden personalizarse permitiendo añadir o eliminar botones en la medida que sea necesario según los propios requerimientos del usuario.

Plug-in *Aplicación que complementa a otro programa informático, agregando una funcionalidad adicional o una nueva característica al software. Normalmente es ejecutado mediante el software principal, con el que interactúa a través de una cierta interfaz.*

CAPÍTULO 1
Introducción

1.1 Motivación

Hoy en día existen infinidad de programas de gran relevancia que necesitan garantizar un correcto funcionamiento. Para ello, necesitamos poder comprender y depurar dicho programa, y así facilitar la resolución de los posibles fallos encontrados. Por ello, es habitual utilizar lenguajes formales que permitan especificar de forma precisa distintos conjuntos de problemas, que en lenguajes imperativos como C o Java pueden ser bastante costosos de describir.

Maude es un lenguaje de programación basado en la lógica de reescritura propuesta originalmente por José Meseguer [1], que permite definir formalmente lo que se quiere expresar en una manera muy abstracta, sin dar importancia a la estructura o la implementación, describiéndola de igual manera a los postulados matemáticos y ecuaciones. De esta forma, partiendo de la especificación entendida como un conjunto de hechos o afirmaciones, se pueden derivar nuevas propiedades del objeto especificado.

Se han definido una amplia diversidad de herramientas formales para el lenguaje Maude que se han aplicado satisfactoriamente en multitud de organizaciones y empresas de todo el mundo en aplicaciones tales como modelos de computación, semánticas de lenguajes de programación, arquitecturas distribuidas, redes de Petri, componentes software, demostración automática de teoremas, certificación de programas, verificación de protocolos de comunicación, etc. Por ejemplo, la herramienta Maude-NPA es un instrumento de análisis de protocolos criptográficos que tiene en cuenta muchas de las propiedades algebraicas de los sistemas criptográficos que no están incluidas en otras herramientas. Éstas incluyen la cancelación de la codificación y decodificación, los grupos abelianos (grupos en los que el resultado de aplicar una operación del grupo a dos elementos del dominio no depende del orden en el que están escritos), las potencias y el cifrado homomórfico (un tipo de cifrado en el que una operación algebraica concreta, sobre un texto original, equivale a otra operación algebraica sobre el mismo texto cifrado). Esta herramienta fue creada con el fin de ayudar a encontrar fallos de seguridad o corroborar que un protocolo está libre de ataques. Otro ejemplo son las herramientas Pathway Logic, que facilitan la comprensión de los sistemas biológicos complejos y aceleran el diseño de expe-

rimentos para probar hipótesis sobre sus funciones en vivo. Entre los usos de esta herramienta está la modulación y el análisis de la transducción de señales y redes metabólicas en células de mamíferos.

No obstante, las herramientas basadas en interacciones con la consola del sistema añaden un tiempo extra para la comprensión del programa y no ofrecen el rendimiento y prestaciones de una aplicación de escritorio. El depurador de programas ANIMA esta disponible a través de una aplicación gráfica *online* que interacciona con la consola de Maude, pero a pesar de todos los beneficios que traen consigo las aplicaciones web, como la compatibilidad multiplataforma o un menor uso de memoria, también tiene sus limitaciones como el tamaño máximo del canvas a la hora de dibujar en HTML5, la cantidad de información que se puede intercambiar entre cliente y servidor, las diferentes interpretaciones de los lenguajes según el navegador, versión y sistema operativo como CSS, HTML5, JavaScript, etc. También se ven comprometidas la rapidez de ejecución, el acceso a Internet, la privacidad (dado que un programa será visible a todo aquel que tenga acceso al servidor), etc. Además, en ocasiones, si el tamaño de los objetos que recibimos del intérprete es demasiado grande, corremos el riesgo de sobrecargar el navegador.

Para evitar este inconveniente el objetivo de este trabajo es desarrollar una aplicación de escritorio que permitirá aprovechar toda la potencia que posee el sistema y no sólo la que le brinda el navegador. Por ello, en este trabajo se desarrolla la herramienta de animación y depuración de programas *d*ANIMA (*desktop* ANIMA), la cual proporciona un entorno versátil para el análisis dinámico de ejecuciones en Maude. La herramienta soporta un *program stepper* que permite la ejecución paso a paso de programas así como un *trace slicer* que computa versiones simplificadas de la ejecución de un programa donde se elimina cualquier información irrelevante para un criterio de observación que selecciona la información de interés.

1.2 Aplicaciones *online* vs aplicaciones de escritorio

Una aplicación on-line es una herramienta que los usuarios pueden utilizar accediendo a un servidor web a través de Internet o de una intranet mediante un navegador, es decir, es una aplicación software codificada en un lenguaje soportado por los navegadores web.

En los últimos años, con la evolución de la tecnología y la necesidad de poder acceder a las aplicaciones desde dispositivos móviles con los mínimos requisitos, se han desarrollado millones de aplicaciones web, debido a la independencia del sistema operativo así como a la facilidad para actualizar y mantener aplicaciones web sin distribuir e instalar software en cada dispositivo útil. Esto es posible dado que dichas herramientas son alojadas en la "nube", es decir, se trata de aplicaciones instaladas en servidores web, lo cual permite trabajar desde cualquier ubicación siempre y cuando esté disponible una conexión a la red.

En cambio, una aplicación de escritorio es una herramienta que está instalada en el

propio dispositivo del usuario y que se ejecuta directamente por el sistema operativo. El rendimiento dependerá por tanto de las características del hardware, incluyendo la capacidad de la memoria RAM, la velocidad de lectura y escritura del disco duro, el rendimiento de la tarjeta gráfica, etc.

La siguiente tabla resume las principales ventajas y desventajas de ambos tipos de aplicaciones.

	Aplicaciones *Online*	Aplicaciones de escritorio
Requiere comunicación con el exterior	✓	✗
Menor tiempo de desarrollo	✓	✗
Requiere la instalación del software	✗	✓
Requiere instalación de librerías adicionales	✗	✓
Mayor rendimiento de contenidos audiovisuales	✗	✓
Mayor velocidad de ejecución	✗	✓
Ofrece mayor seguridad	✗	✓
Requerimientos más estrictos en cuanto a dependencias	✗	✓
Actualización sin necesidad de intervención del usuario	✓	✗
Dispone de un mayor espectro de actuación	✗	✓
Bajo consumo de recursos	✓	✗
Ofrecen mayor personalización	✗	✓
Mayor capacidad de usuarios concurrentes	✓	✗

Tabla 1.1: Comparación entre aplicaciones web y de escritorio

CAPÍTULO 2
Preliminares: Tecnologías utilizadas

En esta sección comentaremos brevemente las diferentes tecnologías que hemos utilizado para desarrollar nuestra aplicación ANIMA de escritorio.

2.1 Maude

El lenguaje Maude fue desarrollado por José Meseguer en el SRI (Stanford Research Institute, California), publicado con la licencia de documentación libre de GNU (*GNU General Public License*). Actualmente, está siendo desarrollado bajo la dirección de José Meseguer, por un equipo internacional de investigadores encabezado por Steven Eker perteneciente al SRI y otros centros de investigación y universidades en Europa y los Estados Unidos.

Maude es un lenguaje de programación lógico funcional [10]. Al disponer de las ventajas de los lenguajes de programación funcionales (p. ej., Haskell), podemos basarnos en el uso de funciones matemáticas que no requieren focalizar los cambios de estado a través de la mutación de las variables. Este tipo de lenguajes se caracterizan por estar construidos por definiciones de funciones que, al contrario que las funciones de los lenguajes imperativos, sólo comprueban sus propiedades.

Por otra parte los lenguajes lógicos como es por ejemplo Prolog, se basan en reglas de inferencia para alcanzar una solución. Esto es, dado un problema y una base de conocimientos, formada por hechos y reglas, donde una regla es un implicación o inferencia lógica que deduce nuevo conocimiento, la aplicación de la regla permite definir nuevas relaciones a partir de otras ya existentes. Un hecho es una declaración o proposición cierta o falsa, establece una relación entre objetos y es la forma más sencilla de sentencia.

Cuando un lenguaje es funcional y lógico hablamos que es un lenguaje declarativo multiparadigma.

Una vez descritas las características generales de los lenguajes de los que proviene Maude vamos a centrarnos en él. La especificación y ejecución de un programa Maude combina variables lógicas, cálculo con información parcial y resolución de

restricciones con propiedades ecuacionales para tratar con tipos de datos avanzados, tipificación con géneros ordenados (*order-sorted*), diferenciación entre componentes, funciones y concurrentes, y módulos parametrizados.

Maude permite abordar un conjunto diferente de problemas a los que resolverían usando los lenguajes imperativos comunes como C, Java o Perl. Es una herramienta de razonamiento formal, lo que puede ayudar a verificar que las cosas son "como es debido", y nos muestran por qué no son así, si es el caso. En otras palabras, Maude nos permite definir formalmente lo que entendemos por algún concepto de una manera muy abstracta (sin preocuparnos de cómo se representa la estructura internamente). Sin embargo podemos describir lo que se considera equivalente en relación con nuestra teoría (ecuaciones) y los cambios de estado por los que puede pasar el sistema (reglas de reescritura). Esto es muy útil para validar los protocolos de seguridad y el código crítico.

La lógica ecuacional con membresía (o *"membership equational logic"*) da cuenta del concepto de subsorting (o subtipo) incorporando una variedad de posibles relaciones entre *sorts*. La lógica de reescritura, por su parte, es una lógica relativa al cambio concurrente, que da cuenta naturalmente del concepto de estado y de cómputos concurrentes y no deterministas. Esta lógica constituye un marco general para dotar de semántica a una amplia gama de lenguajes y modelos de concurrencia. Provee, en particular, muy buen soporte a cómputos en el marco de los Objetos Concurrentes [1].

Maude tiene sus raíces en el lenguaje OBJ3 creado por Joseph Goguen [13] que implementa una lógica ecuacional, dado que el sublenguaje de la lógica ecuacional de Maude contiene principalmente a OBJ3 como sublenguaje. La principal diferencia de Maude con respecto a OBJ3 es que Maude le da soporte a una lógica más rica, que extienden la lógica ecuacional de géneros ordenados de OBJ3.

En la aplicación *d*ANIMA utilizaremos Mau-Dev [12], una extensión para desarrolladores del lenguaje Maude que dota a la última versión oficial Maude 2.7 con una serie de nuevas y útiles operaciones en el meta-nivel para desarrolladores implementadas nativamente en C++. Mau-Dev es totalmente compatible con Maude 2.7 y mantiene la eficiencia de todas las operaciones estándar en el meta-nivel de Maude y sus comandos.

2.2 Java

Java es un lenguaje de programación creado por James Gosling y Bill Joy , este procede de otro lenguaje llamado Oak diseñado para la creación de aplicaciones interactivas para la televisión.

Java es un lenguaje de programación concurrente y orientado a objetos, que fue diseñado con la idea de que sus programas se ejecutaran independiente de la má-

quina en la que se ejecute y seguro para utilizarlo en red, consiguiendo menores dependencias de implementación. Java consigue la independencia de ejecución de la plataforma gracias a su máquina virtual de Java (JVM), capaz de interpretar el código binario que genera el compilador (*bytecode* Java), y de ahí el axioma de Java: *"write once, run anywhere"* [14].

Una de las características más relevantes de este lenguaje es que es compilado e interpretado como hemos podido introducir al hablar de la JVM. Gran parte de su éxito fue debido a que elimina el trabajo con punteros al programador, y accediendo a las variables a través de referencias, la comprobación estricta de tipos en el momento de compilación evitando posibles errores.

Hay que mencionar, además, que Java implementa la técnica de recolección de basura (*automatic garbage collection*) que evita en gran medida las fugas de memoria. De esta forma, cuando no quedan referencias al objeto en cuestión, el recolector de basura de Java borra el objecto, liberando así la memoria que éste ocupaba.

Entre las múltiples librerías de Java para crear una UI (Interfaz de Usuario) hemos elegido para nuestro proyecto la que nos ofrece JavaFX.

2.3 Maven

Maven es una herramienta de gestión y construcción de proyectos software creada por Jason van Zyl, de Sonatype, en 2002, a partir de la creación de un documento basado en el modelo de objeto de proyecto (POM) [8]. Mediante este artefacto, se puede gestionar la compilación de un proyecto, generar informes y documentación de un proyecto a partir del documento POM, en el cual estará la estructura que deberá tener el proyecto y las dependencias de otras librerías en caso de que las haya.

El principal objetivo de Maven es facilitar a los desarrolladores a comprender el estado completo tras un ciclo de desarrollo en el menor período de tiempo posible. De esta forma, se plantean las siguientes ideas para lograr alcanzar esta meta:

- Proporcionar un sistema de construcción uniforme.
- Ofrecer directrices para la construcción de código utilizando las mejores prácticas, consiguiendo proporcionar de la calidad del proyecto.
- Simplificar el proceso de construcción.
- Realizando de forma transparente la migración de una librería a otra o a su actualización.

De los planteamientos indicados anteriormente Maven consigue ofrecer las directrices gracias a que la herramienta recopila los principios actuales para el desarrollo de mejores prácticas, procurando que su aplicación sea fácil aplicarlos, obteniendo

la mejor forma de guiar el proyecto en la dirección deseada.

La especificación, ejecución y presentación de informes de las pruebas unitarias, por ejemplo, son parte del ciclo de compilación normal que utiliza Maven. Actualmente nos indican que las mejores prácticas para realizar las pruebas unitarias las podemos simplificar en los siguientes puntos:

- Mantener el código fuente de las pruebas separado del código fuente de desarrollo, pero manteniendo ambos en paralelo.
- Seguir la convención para la definición de nombres para los casos de prueba con el fin de localizar y ejecutar las pruebas.
- Permitir la creación y personalización de los casos de prueba en el mismo entorno de desarrollo. Esto permite ejecutar dichas pruebas manualmente o antes de compilar el proyecto.

Maven proporciona una cuantiosa cantidad de información útil del proyecto, en su mayoría a través de su POM, y también generada a partir de las fuentes del proyecto. Entre todas las opciones que ofrece esta herramienta podemos destacar:

- Cambiar el documento donde se registran las trazas del programa directamente desde el sistema de control de versiones de código.
- Cruzar las distintas referencias de las fuentes.
- Listar correos electrónicos.
- Listar dependencias de librerías.
- Generar informes sobre las pruebas unitarias, incluyendo la cobertura del código.

En resumen, cuando se desarrolla código en Java, el uso de librerías externas es habitual para complementar o facilitar el desarrollo.

2.4 JavaFX

JavaFX es una familia de productos y tecnologías de Sun Microsystems, adquirida por Oracle, para la creación de *Rich Internet Applications* (RIAs), es decir, aplicaciones web que tienen las características y capacidades de aplicaciones de escritorio.

Contiene un conjunto de tecnologías y herramientas para el diseño, desarrollo y despliegue de la aplicación. Es de contenido expresivo, lo que facilita ser visualizado en navegadores de Internet y equipos con pantalla. Además ofrece alta fidelidad en audio y vídeo, textos enriquecidos, gráficos vectoriales, animaciones y servicios Web.

CAPÍTULO 3

Una aplicación de escritorio para el análisis dinámico de programas Maude

En este apartado se describe en qué consiste la aplicación desarrollada y el camino seguido para implementar dicha aplicación *d*ANIMA.

3.1 Arquitectura del sistema

La arquitectura del sistema implementada en nuestra aplicación está basada en las tradicionales arquitecturas donde sólo se requiere el propio sistema informático garantizando su ejecución en un entorno aislado.

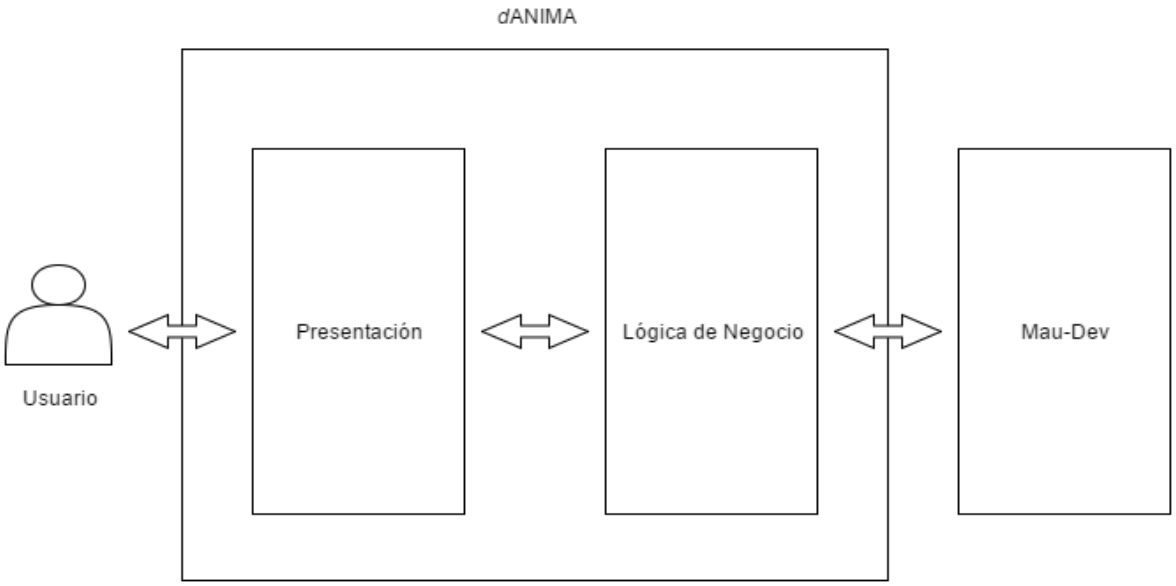

Figura 3.1: Arquitectura del Sistema

Como se muestra en la Figura 3.1, las diferentes capas de la arquitectura de la aplicación *d*ANIMA son:

- *Presentación:* La capa de presentación abarca las vistas, los controladores de las vistas, los componentes gráficos y la comunicación con la capa de la lógica de negocio.
- *Lógica de negocio:* La capa de la lógica de negocio se encarga de convertir los componentes gráficos de la capa de presentación en objetos Java. Para la creación de dichos objetos necesitamos convertir los objetos JSON que nos devuelve el intérprete en objetos Java. Posteriormente, en la sección de diagrama de clases profundizaremos en la lógica de negocio, donde se explica la funcionalidad interna.
- *Intérprete Mau-Dev:* Mau-Dev es un intérprete que recibe las ordenes por parte de la capa de la lógica de negocio. Una vez nos da su respuesta, cerramos la comunicación con él. De esta forma, cada vez que el usuario realice una acción el intérprete iniciará una nueva conexión.

3.2 Funcionalidades

Una aplicación para el análisis dinámico de programas Maude debe aportar varias funcionalidades necesarias, ya que su finalidad es mejorar la experiencia del usuario respecto a la utilización del intérprete como herramienta de primer contacto.

3.2.1. Editor de código

El editor de código a través de la pantalla principal brinda la posibilidad de escribir código en lenguaje Maude, el cuál permite detectar las palabras clave propias

del lenguaje, así como numerar las líneas escritas. De esta manera, si tras interactuar con el intérprete Mau-Dev se encuentra un error de sintaxis, se avisará al usuario dándole información precisa de la línea en la cual puede haber cometido el fallo.

3.2.2. Importar código en Maude

Otra funcionalidad de la pantalla principal es la opción de importar código en Maude, que podrá ser creado por el propio usuario y guardado en un fichero de texto. Tras su importación se detectarán las palabras clave y se marcarán con el estilo indicado.

3.2.3. Cambiar el color asociado a un estado

Se ofrece la posibilidad de cambiar el color asociado al estado del nodo a criterio del usuario final.

3.2.4. Arrastrar y soltar el árbol

Se ofrece la posibilidad de arrastrar y soltar todos los nodos que se hayan desplegado en un momento dado en el árbol.

3.2.5. Expandir un nodo

Una de las funcionalidades más importantes de la aplicación es expandir un nodo junto con la acción de mostrar los nodos no normalizados tras su anterior expansión. Dado que se trata de una de las acciones más importantes de la aplicación, se han previsto dos maneras para poder ejecutar dicha acción, ya sea por botón o en un menú desplegable informando de la acción.

3.2.6. Plegar un nodo

También tenemos la acción de plegar el nodo seleccionado siempre y cuando se haya expandido ya. Si no fuera el ultimo nodo expandido de esa rama, plegaremos todos los nodos que cuelguen de él, ahorrando al usuario la ardua tarea de plegar los nodos uno a uno o tener que volver atrás para recuperar el estado inicial.

3.2.7. Expandir un nodo dada una profundidad

De manera semejante, aportamos la funcionalidad de poder expandir un nodo con un sólo movimiento profundizando en el árbol de cómputos tanto como se quiera. Los nodos hijos se pueden expandir hasta llegar al límite fijado.

Example 3.2.1. Elegimos el problema "*Blocks World*" dado que es uno de los dominios de planificación más famosos en la rama de inteligencia artificial. El problema

```
mod BLOCKS-WORLD is inc INT .
  sorts Block Prop State .
  subsort Prop < State .
  ops a b c : -> Block .
  op table : Block -> Prop . *** block is on the table
  op on : Block Block -> Prop . *** block A is on block B
  op clear : Block -> Prop . *** block is clear
  op hold : Block -> Prop . *** robot arm holds the block
  op empty : -> Prop . *** robot arm is empty
  op _&_ : State State -> State [assoc comm] .
  op size : Block -> Nat .
  vars X Y : Block .

  eq [sizeA] : size(a) = 1 .
  eq [sizeB] : size(b) = 2 .
  eq [sizeC] : size(c) = 3 .

  rl [pickup] : clear(X) & table(X) => hold(X) .
  rl [putdown] : hold(X) => empty & clear(X) & table(X) .
  rl [unstack] : empty & clear(X) & on(X,Y) => hold(X) & clear(Y) .
  crl [stack] : hold(X) & clear(Y) => empty & clear(X) & on(X,Y) if
      size(X) < size(Y) .
endm
```

Figura 3.2: Módulo BLOCKS-WORLD

empieza planteando un estado inicial, en el cual tenemos una mesa con varios cubos o bloques y debemos construir torres con esos bloques con la restricción de no poder mover más de un bloque al mismo tiempo, ya que el encargado de mover los cubos es un único brazo robot. Tampoco podemos apilar un bloque encima de otro bloque cuyo tamaño sea mayor que el bloque ya apilado o mover los bloques que se encuentren debajo de otros. El brazo del robot puede elegir entre dejar el cubo en la mesa o encima de otro bloque[2].

Llegados a este punto, procedemos a explicar el módulo *Blocks World* especificando qué ecuaciones y reglas se pueden aplicar para realizar las transiciones entre estados.

Como podemos observar en la Figura 3.2, tenemos tres bloques distintos de bloques: a, b y c. Mediante operadores (*op o ops*) definimos los diferentes tamaños para cada tipo de bloque (*size*) o propiedades básicas para especificar estados de los bloques o del brazo del robot: *table, on, clear, hold, empty*. Los estados son modelados mediante listas asociativas y conmutativas de propiedades de los bloques y del brazo del robot.

Finalmente, por medio de las cuatro reglas: *pickup, putdown, unstack y stack*, se describe el comportamiento del sistema mediante el control del brazo del robot. Centrándonos en la información que nos aportan las reglas tenemos:

- *Pickup:* Esta regla describe la forma en la que el brazo del robot coge un bloque de la mesa.
- *Putdown:* Esta regla describe la manera en la que el brazo del robot deja el bloque sobre la mesa.
- *Unstack:* Esta regla especifica la acción que hace el brazo del robot para coger un bloque que está encima de otro bloque o de una pila de bloques.
- *Stack:* Esta regla especifica la acción que hace el brazo del robot para dejar el bloque encima de otro bloque o una pila de bloques.

El módulo BLOCKS-WORLD es un ejemplo muy sencillo para explicar esta funcionalidad dado que se puede ver claramente el resultado final.

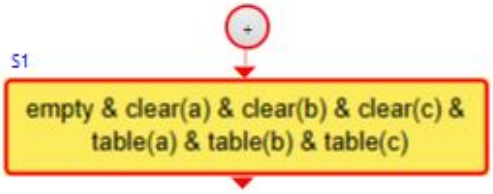

Figura 3.3: Estado inicial del ejemplo Blocks-World

Por un lado vemos en la Figura 3.3 como el usuario inicialmente tiene un árbol con dos nodos, el nodo raíz y su forma normalizada. Donde el estado inicial plantea que el brazo del robot está vacío y los bloques están dispuestos directamente sobre la mesa.

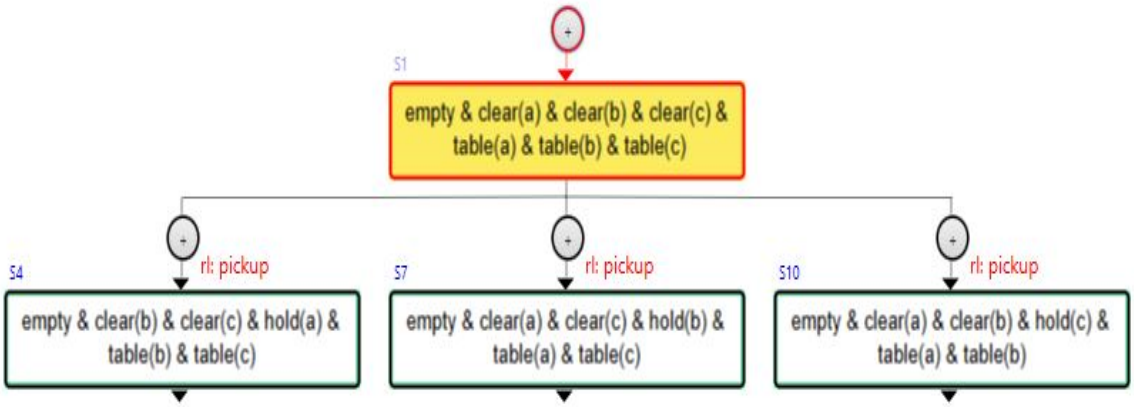

Figura 3.4: Árbol con profundidad 1

El resultado de expandir el nodo sería el árbol que vemos en la Figura 3.4, donde tendremos tres nodos normalizados (S4, S7 y S10) y seis nodos intermedios (S2, S3, S5, S6, S8 y S9). En este caso, los nodos normalizados se consiguen cuando el brazo del robot tiene un bloque y el resto de bloques están en la mesa, y para esto, se debe aplicar la regla *pickup*.

No obstante, si el usuario quiere visualizar el árbol a una determinada profundidad, mediante esta funcionalidad expandirá los nodos de todos los niveles hasta llegar a la profundidad deseada.

Tras la acción de expandir dos veces el sub-árbol, obtiene los nodos normalizados (S4, S7 y S10) de profundidad uno y (S13, S16, S19, S22, S25, S28, S31, S34, S37, S40, S43 y S46) de profundidad dos; y los nodos intermedios (S2, S3, S5, S6, S8 y S9) de profundidad uno y (S11, S12, S14, S15, S17, S18, S20, S21, S23, S24, S26, S27, S29, S30, S32, S33, S35, S36, S38, S39, S41, S42, S44 y S45) de profundidad dos.

3.2.8. Mostrar la información del estado

Añadimos la posibilidad de observar la información del estado en el que se encuentra el nodo seleccionado.

3.2.9. Mostrar la información de la transición

Finalmente, también añadimos la posibilidad de observar la información sobre la transición que se ha producido para llegar a dicho estado, mostrando la ecuación normalizada, las sustituciones y la posición.

3.3 Otros detalles de implementación

En esta sección profundizaremos en los detalles referentes a la implementación.

3.3.1. Interfaz gráfica de usuario

Una interfaz de usuario es un medio con el cuál poder comunicarse con un dispositivo que trata de facilitar toda interacción entre el usuario y el equipo. Dado que el usuario puede en todo momento hacer estas interacciones mediante el intérprete, nosotros queremos intermediar y mejorar el entendimiento de dicho funcionamiento.

Nuestra aplicación permite introducir código en lenguaje Maude y su estado inicial o bien elegir entre módulos o programas previamente introducidos a modo de ejemplo y empezar así la obtención y análisis de las trazas.

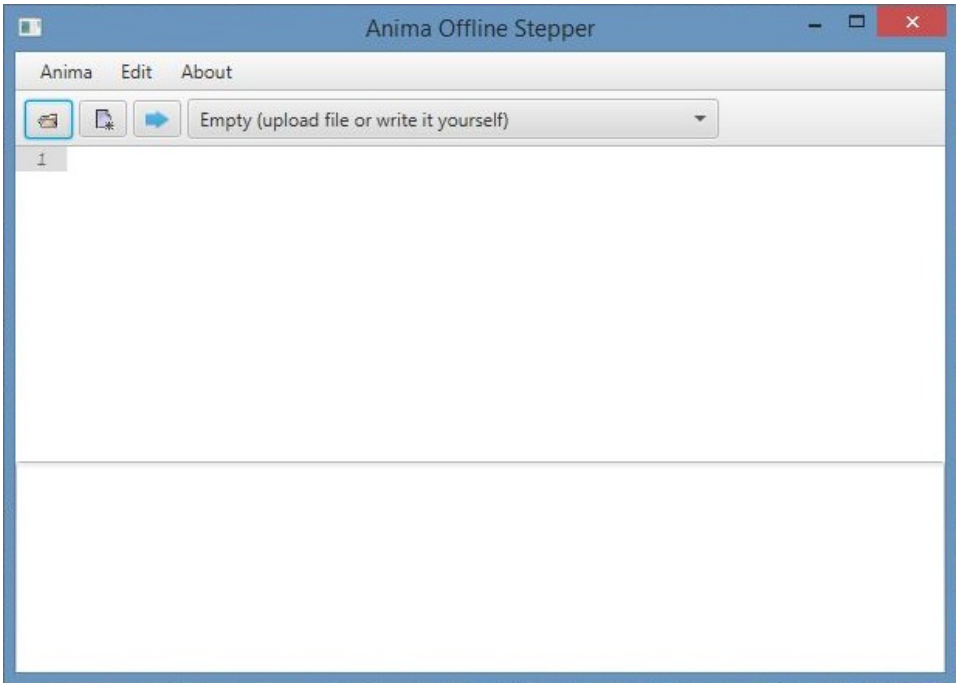

Figura 3.5: Pantalla Principal de la aplicación

3.3.2. Editor de código

En la pantalla principal se ofrece un área donde el usuario puede escribir cualquier módulo en lenguaje Maude que se desee depurar, como haría en cualquier editor de código externo (p. ej.: Sublime Text, Notepad++, Emacs, etc). No sólo es posible introducir código en una región de dicha pantalla sino también poder aplicarle cambios de estilo a las palabras clave, propias del lenguaje de programación (*keywords*). Para el programador resulta mucho más sencillo escribir con la apariencia de editor de código inteligente, ya que la creación de estilos por palabras clave permite visualizar de forma esquemática el contenido y así, poder encontrar un error de sintaxis rápidamente a diferencia de un editor de texto plano simple.

```
1  fmod BANK-INT+ID is inc INT .
2    sort Id .
3  endfm
4
5  view Id from TRIV to BANK-INT+ID is
6    sort Elt to Id .
7  endv
8
9  fmod BANK-EQ is
10   inc BANK-INT+ID .
11   pr SET{Id} .
12
13   sorts Account PremiumAccount Status Msg State .
14   subsort PremiumAccount < Account .
15   subsorts  Account Msg < State .
16
17   var ID : Id
```

Figura 3.6: Área de código con estilos

La Figura 3.6 sirve para ilustrar lo dicho, donde podemos ver cómo se han marcado en un color celeste las palabras: **“fmod”**, **“is”**, **“inc”**, **“sort”**, **“endfm”**,

“view”, **“endv”**, **“pr”**, **“subsort**’ y **“subsorts”**, así como los tokens: **“ . ”** y **“<”**.

También podemos observar que en la Figura 3.6 marcamos las lineas que va escribiendo el usuario, de forma que si éste introdujera un error en la sintaxis inmediatamente se le informa dónde ha cometido el fallo, de forma que él podría ir directamente a la linea en cuestión.

Para poder aportar estas ventajas al usuario final de la aplicación, nos apoyamos en una herramienta hecha en JavaFX llamada RichTextFX creada por Tomas Mikula [**?**]. Esta herramienta permite asignar clases de estilo para rangos de texto, previamente definidas dichas clases en una hoja de estilos.

Dentro de la librería RichTextFX, nosotros usaremos la clase CodeArea, una variante de StyleClassedTextArea que utiliza una fuente de ancho fijo de forma predeterminada, por lo que es una base ideal para editores de código fuente. Estas fuentes fueron creadas con licencias de software libre: BSD 2-Clause License y GPLv2.

3.3.3. Diagrama de clases

En primer lugar, necesitamos guardar la información relevante que se obtiene tras las interacciones con el intérprete. Para ello, creamos dos clases principales: *Tree* y *NodeAnima*. En segundo lugar, creamos unas clases auxiliares que ayudarán a separar todas las partes importantes del nodo: *Term*, *Rule* y *Substitution*. En ambos grupos de clases recae la lógica de la aplicación, formando el esqueleto básico de lo que nosotros llamaremos *Árbol de Anima*.

En último lugar, tenemos dos clases encargadas de mostrar gráficamente la estructura creada mediante los dos grupos anteriores. Estas clases son *NodeGraphicAnima* y *TreeGraphic*, donde *NodeGraphicAnima* es una subclase que hereda de la clase *NodeAnima*.

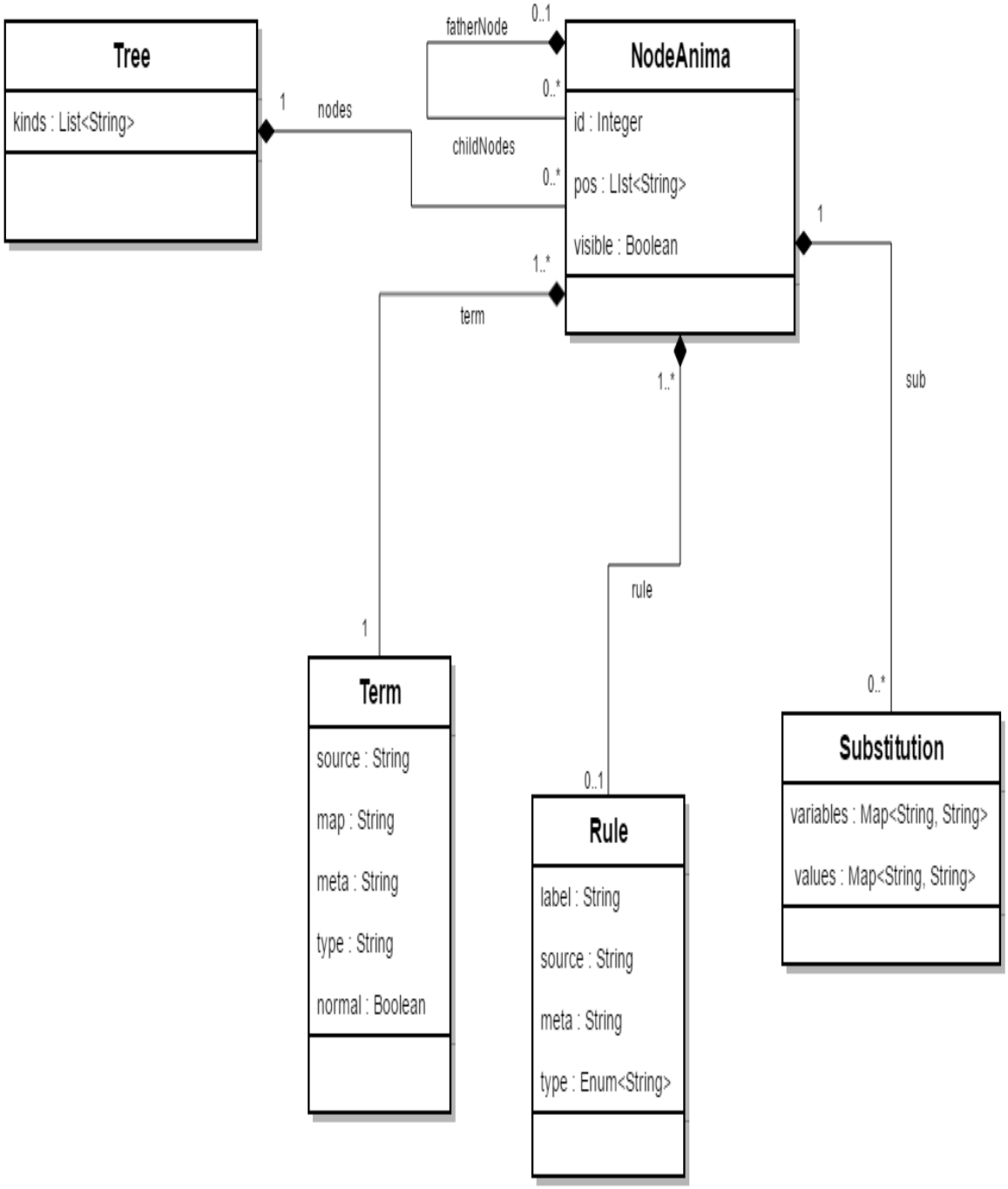

Figura 3.7: Diagrama de clases *d*ANIMA

A continuación procederemos a explicar las distintas clases que aparecen en nuestro diagrama de clases:

- *Tree*: Esta clase contiene los nodos totales que tiene el árbol y los tipos de valores.
- *NodeAnima*: Esta clase contiene toda la información relevante del nodo. Nos guardamos tanto su número identificador en el árbol, la posición, la visibilidad que dependerá de si es un nodo normalizado y el último en la lista de nodos obtenidos en una transición, teniendo en cuenta que puede obtenerse uno solo.
- *Term*: Esta clase contiene la información del nodo, el tipo y si es normalizado.
- *Rule*: Esta clase encapsula la información sobre la regla u ecuación aplicada y el tipo de regla u ecuación aplicada.
- *Substitution*: Esta clase contiene las variables y valores de la sustitución.

3.3.4. Nodo *d*ANIMA

Primero optamos por encapsular los componentes necesarios para mostrar la información del nodo: un cuadro de texto, dos botones, líneas, unos polígonos creando la forma de las flechas, etiquetas y los contenedores necesarios para crear una estructura parecida a la actual aplicación ANIMA, respetando así mismo el diseño original:

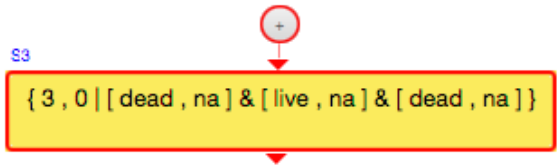

Figura 3.8: Nodo de la aplicación *d*ANIMA plegado

Como podemos observar en la Figura 3.8, tenemos dos botones, uno con el símbolo de suma y otro con una S (*state*) y la numeración del nodo en el árbol. Si pulsamos sobre el botón con el signo de la suma obtenemos todos los nodos no normalizados tras interactuar con el intérprete. El botón cambiará de signo convirtiéndose en resta, dando la posibilidad de volver a pulsarlo y esconder todos los nodos no normalizados antes del nodo asociado. En cambio, cuando el usuario pulsa sobre la etiqueta del estado, nos quedamos con el número y efectuamos una petición al intérprete para que nos dé los nodos correspondientes a dicha traza.

3.3.5. Posicionamiento de los nodos *d*ANIMA

La versión web de ANIMA está programada utilizando el algoritmo de posicionamiento de nodos para árboles generales (A Node-Positioning Algorithm for General

Trees) de John Q. Walker II. Dicho algoritmo determina la posición de los nodos para cualquier árbol general, especificando la posición x e y del nodo, que minimiza la anchura del árbol, dado que en un árbol general no hay límite de nodos hijos por cada nodo, a diferencia de los árboles binarios y ternarios. Este algoritmo se ejecuta en un tiempo O(N), donde N es el número de nodos en el árbol y aborda el problema de la elaboración de estructuras de árbol.

Este algoritmo aborda el problema de la elaboración de estructuras basadas en árboles, ya que se utilizan habitualmente para representar una estructura organizada jerárquicamente. En el área informática, los árboles se utilizan para la búsqueda de información, la compilación de programas o para los sistemas de bases de datos. La elaboración de esquemas en árbol ayuda a entender las relaciones jerárquicas mucho mejor que si la información estuviera distribuida de forma lineal.

Volviendo al tema que nos ocupa, dado que el objetivo de nuestra aplicación es mejorar la experiencia del usuario, debemos distribuir la información tras la respuesta del intérprete Mau-Dev por la pantalla de forma intuitiva y clara. Para ello, se requiere elegir las posiciones de los nodos resultantes dándoles unas coordenadas específicas y un tamaño óptimo.

En la versión de Anima *online*, se utiliza el algoritmo mencionado anteriormente, teniendo en cuenta que está desarrollada en un lenguaje de programación interpretado, es decir, requiere de un programa auxiliar (el intérprete), que descifra los comandos según sea necesario. Uno de los inconvenientes de dicho lenguaje de programación es la necesidad de añadir *plugins* y *frameworks* para hacer más fácil la elaboración de código. Por eso, la aplicación original se diseñó de manera más rápida siguiendo el algoritmo de posicionamiento de nodos para árboles generales con el fin de recalcular todas las posiciones de los nodos tras cualquier acción del usuario dado que resulta mucho más complicado diseñarlo para que el renderizado interno del navegador lo re-calcule.

Como hemos mencionado, esta tarea se consigue mediante un algoritmo de posicionamiento de nodos que calcula la coordenadas x e y para cada nodo del árbol. La rutina de representación puede entonces utilizar estas coordenadas para dibujar el árbol. Un algoritmo de posicionamiento de nodos debe abordar dos cuestiones fundamentales. En primer lugar, el dibujo resultante debe ser estéticamente agradable. En segundo lugar, el algoritmo de posicionamiento de nodos debe ser capaz de reservar el espacio perteneciente a cada nodo. Estas dos características pueden parecer triviales, pero conseguir las dos a la vez puede resultar una tarea más difícil.

Dicho lo anterior y considerando que nuestro código está escrito en Java utilizando un framework integrado en sus propias librerías como es JavaFX, esto ya nos permite encapsular un componente, previamente creado e inicializado, dentro de unos contenedores denominados 'layouts', donde cada uno tiene una orientación distinta.

Estos paneles son unos contenedores que se usan para organizar de forma flexible y dinámica todos los accesos y controles de la interfaz de usuario dentro de un escenario gráfico de una aplicación JavaFX. Cuando una ventana cambia de tamaño, el panel que la ventana posea en su interior actualizará sus coordenadas y cambiará

su tamaño, provocando que todos los nodos que posea éste, también se actualicen de igual forma.

Llegados a este punto, procedemos a explicar los diferentes *layouts* que posee el *framework* y cómo hemos podido beneficiarnos de sus múltiples ventajas.

Uno de los *layouts* más comunes es el BorderPane, que permite distribuir las distintas componentes entre sus 5 posiciones y así evitar que se junten tras el renderizado.

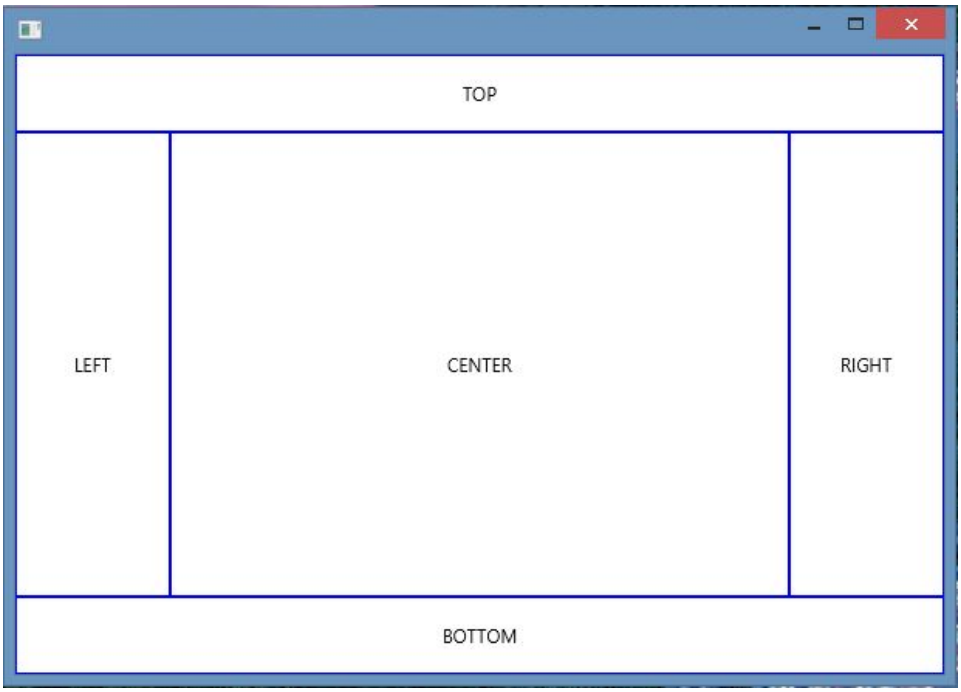

Figura 3.9: Border Layout

En la Figura 3.9, observamos las 5 zonas mencionadas anteriormente. Habitualmente se suele utilizar la zona de arriba para un Menú o bien una barra de herramientas, también llamada *Toolbar*, siendo estos los que permiten simplificar la navegabilidad (ofrecer ayuda, étc).

Hay que mencionar, además, que la parte central es la más utilizada, ya que es donde se suele encontrar el núcleo de la aplicación. En nuestro caso, es donde tenemos todos los nodos del árbol y nos permite separar las distintas partes de nuestra aplicación de forma que si movemos los nodos mediante el *Drag and Drop* no moverá las otras partes de la aplicación.

Una de las ventajas que nos aporta utilizar *layouts*, es que podemos encapsularlos dentro de otros *layouts* tantas veces como deseemos. Por lo tanto, si desplazamos un *layout* exterior, todos los *layouts* interiores se moverán con él, recalculando internamente su nueva posición.

Otros tipos de *layouts* que ofrecen las librerías de JavaFX y hemos podido utilizar son:

- StackPane: Este *layout* distribuye los componentes añadidos en forma de pila; es decir, el último nodo insertado estará en la parte superior del contenedor y el primer nodo en la zona inferior. Dependiendo del tamaño de los nodos superiores se verán más o menos los nodos inferiores.

  Además, dicho contenedor intentará cambiar el tamaño de cada componente para llenar su área de contenido. Si el componente no pudiera ser dimensionado para llenar el contenedor principal (ya sea porque es de tamaño variable o su tamaño máximo lo impide), entonces se alinea dentro de la zona utilizando la propiedad de alineación, que de forma predeterminada será centrada. No obstante, podemos alinear cada objeto a nuestra conveniencia.

- Pane: Este *layout* se puede utilizar para posicionar los nodos en coordenadas absolutas. Lo cual, suele ser para usos específicos, como el posicionamiento de gráficos e imágenes.

- HBox: Este *layout* es uno de los contenedores más básicos que posee la librería JavaFx. Su función es repartir los nodos que incluye en una única fila horizontal. Este panel se suele utilizar en cooperación con otros gestores de diseño para crear diseños más complejos.

- VBox: Este es otro de los contenedores más básicos que posee la librería JavaFx. Su cometido es organizar los nodos que engloba en una única fila vertical. Este panel, al igual que un HBox, se suele utilizar junto con otros contenedores de mayor magnitud para fines de diseño.

- Group: Este *layout* agrupa todos los nodos que contenga exclusivamente. Por ello, sólo contendrá los nodos sin incluir el espacio libre que haya entre ellos en la ventana. Además, este panel no es de tamaño variable, lo que significa que no es manejado por su nodo padre dentro del escenario gráfico.

Dicho lo anterior podemos explicar como hemos hecho uso de dichos contenedores.

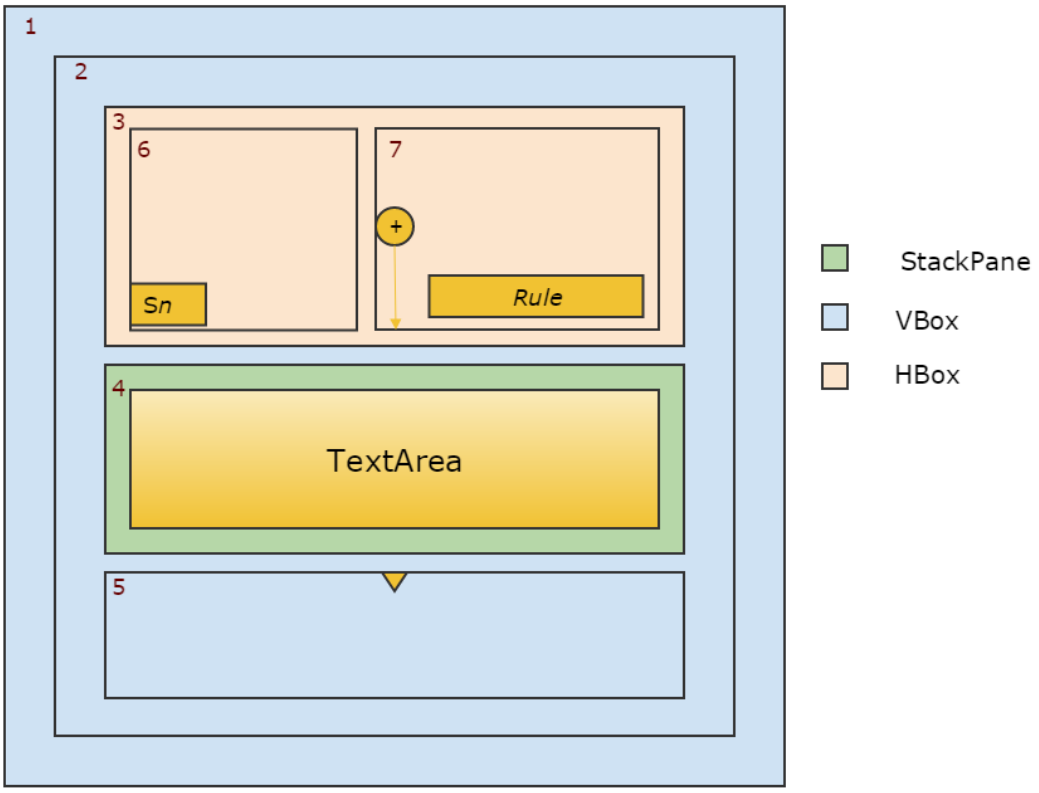

Figura 3.10: Contenedores del nodo *d*ANIMA

Como vemos en la Figura 3.10, los contenedores utilizados para creación del nodo *d*ANIMA son: *StackPane, VBox y HBox*. Todos ellos los hemos numerado para poder referirnos a ellos.

- Contenedor 1: Este contenedor es se encarga de unir el nodo con sus nodos hijos.
- Contenedor 2: Este layout es se encarga de unir el nodo con los nodos obtenidos durante la transición.
- Contenedor 3: Este contenedor encapsula dos partes importantes del nodo, los contenedores que contienen los botones para expandir y plegar/desplegar los nodos de la transición, y la regla aplicada.
- Contenedor 4: Este layout se encarga de asegurar que el componente TextArea, componente que muestra la información del estado, ocupe la última posición de la pila y por tanto, estar siempre al frente.
- Contenedor 5: Este contenedor puede contener una flecha o bien unas lineas que le lleven a los nodos hijos.

- Contenedor 6: Este layout contiene el botón que se encarga de expandir dicho nodo.
- Contenedor 7: Este contenedor contiene el botón de pliega/despliega los nodos obtenidos en la transición y la regla aplicada.

En resumen, los nodos de la aplicación *d*ANIMA se asemejan a las tradicionales muñecas rusas *matrioshkas*, el primer nodo normalizado contendrá a todos los que se vayan creando.

3.3.6. Codificación de la aplicación

A continuación vamos a explicar las partes más importantes de la codificación realizada en la aplicación final, en la que destacaremos aquellas clases de mayor relevancia y algunas de sus funciones. De esta manera nos centraremos en las siguientes clases:

- *NodeAnimaGraphic:* Esta clase gestiona la creación de los nodos, el pliegue y despliegue de los nodos hijos y de los nodos resultantes de la transición y actualiza los colores de los nodos dependiendo del camino seguido.

  Como podemos observar en la Figura 3.11, el primer paso es obtener los nodos procedentes de la expansión del nodo seleccionado. Tras este primer paso, recorremos la lista de nodos y para cada uno buscamos el nodo normalizado, la regla aplicada en esa transición y creamos un nuevo nodo. Además actualizamos los colores de los nodos y el camino seguido.

```
private Integer expandNode() {

   List<NodeAnima> nodeList =
       DataProcessing.getNodesFromExpand(this.getId());

   ...

   for (int i = 0; i < nodeList.size(); i++) {
      NodeAnima normalizedNode =
          DataProcessing.getNormalizedNode(nodeList.get(i).getId());
      String regla = DataProcessing.getRule(nodeList.get(i).getId(),
          normalizedNode.getId());
      NodeAnimaGraphic newChildrenNode = new
          NodeAnimaGraphic(normalizedNode.getId(),
          normalizedNode.getTerm(),
         normalizedNode.getPadre(), normalizedNode.getChildNodes(),
            normalizedNode.getRule(), normalizedNode.isVisible(), this);
      TreeGraphic.visualNodes.put(newChildrenNode.getId(),
          newChildrenNode);

      ...

      ChangeListener<Object> listener = (obs, oldValue, newValue) ->
          updateLine(childrenToFather, lineFatherToChildren,
          newChildrenNode.getVboxChildrenNodes());

      lineFatherToChildren.boundsInLocalProperty().addListener(listener);
      lineFatherToChildren.localToSceneTransformProperty()
      .addListener(listener);
      newChildrenNode.getVboxChildrenNodes().boundsInLocalProperty()
      .addListener(listener);
      newChildrenNode.getVboxChildrenNodes().localToSceneTransformProperty()
          .addListener(listener);

      ...
   }

   ...

   updatePathNodes(this);

}
```

Figura 3.11: Codificación de la acción de expandir un nodo

- *DataProcessing:* Esta clase se encarga de aportar los métodos necesarios para representar la información obtenida del intérprete y transformarla al formato que requieren nuestros objetos. También proporciona métodos para obtener los nodos resultantes de una transición y la regla aplicada en la transición, entre otras funcionalidades.

La información obtenida viene encapsulada dentro de un objeto JSON que tratamos para recoger la información que necesitamos y guardar en nuestros objetos Java.

Como se puede apreciar en la Figura **??** y en la Figura 3.12, tratamos la información recibida por el intérprete, comprobamos que el árbol no esté creado ya, por lo tanto sea el nodo inicial; y comenzamos a guardar los datos relevantes en nuestra aplicación.

```
//Reemplazamos la marca "ELP-DQ" que deja el intérprete para separar
    campos por comillas.
String quote = "\u005c\u0022", textWithoutQuotes =
    text.trim().substring(1,text.trim().length() - 1);
String textWithoutELP_DQ = textWithoutQuotes.replaceAll("ELP-DQ", quote);

JSONObject jsonObject = new JSONObject(textWithoutELP_DQ);
```

```
//Si no contiene el término "expands", creamos el árbol y nodo raíz.
if (!textWithoutELP_DQ.contains("expands")) {
   //Creamos el árbol
   tree = new Tree();
   //Creamos el nodo raíz.
   NodeAnima rootNode = new NodeAnima();
   rootNode.setId(totalNodes);
   ...
   //Guardamos la información del estado y comprobamos que no sea
       normalizado.
   rootNode.setTerm(parseTerm(jsonObject.getJSONObject("term")));
   if (rootNode.getTerm().isNormal()) {
      if (jsonObject.has("trace")) rootNode.setVisible(false);
      else rootNode.setVisible(true);
   }
   ...
   //Si tiene más nodos tras él, no esta normalizado. Recorremos la lista.
   if (jsonObject.has("trace")) {
      JSONArray jsonTrace = jsonObject.getJSONArray("trace");
      //Nos guardamos el nodo previo para asociarlo como nodo padre.
      NodeAnima previousNode = rootNode;

      for (int counterTraces = 0; counterTraces < jsonTrace.length();
          counterTraces++) {
         //Creamos un nodo con la información de la regla aplicada, el
             término, su padre, su posición, su id y comprobamos si esta
             normalizado.
         NodeAnima newNodeTrace = new NodeAnima();
         newNodeTrace.setRule(parseRule(jsonTrace.getJSONObject(
             counterTraces ).getJSONObject("rule")));
            newNodeTrace.setTerm(parseTerm(jsonTrace.getJSONObject(
                counterTraces ).getJSONObject("term")));
            newNodeTrace.setId(totalNodes);
            newNodeTrace.setPadre(previousNode);
            newNodeTrace.setPos(jsonTrace.getJSONObject(counterTraces
                ).getString( "pos" ));
         if (jsonTrace.getJSONObject(counterTraces).getJSONArray("sub"
             ).length() > 0) {
               newNodeTrace.setSub(parseSubstitution(jsonTrace
               .getJSONObject(counterTraces ).getJSONArray("sub")));
         }
         if (jsonTrace.getJSONObject(counterTraces).getJSONObject("term"
             ).getBoolean( "normal" ) && counterTraces ==
             jsonTrace.length() - 1)
            newNodeTrace.setVisible(true);
         else newNodeTrace.setVisible(false);

         //Anyadimos el nodo al árbol y actualizamos referencias.
         tree.getNodes().put(totalNodes, newNodeTrace);
         totalNodes++;
         previousNode = newNodeTrace;
      }
      ...
```

Figura 3.12: Codificación de la creación del árbol

3.4 Restricciones de uso

La aplicación *d*ANIMA es una aplicación portable, es decir, no requiere de una instalación previa. No obstante, antes de hacer uso de la aplicación serán necesarias tres cosas a tener en cuenta:

- *Permisos de ejecución*: Para que pueda ejecutarse en un entorno Unix, debemos dar permisos de ejecución al intérprete Mau-Dev. Los ficheros en concreto son: maude.darwin (Mac OS X) y maude.linux (Linux).
- *Cygwin*: Para que pueda ejecutarse en el sistema operativo Windows, debemos instalar la herramienta Cygwin.
- *Máquina Virtual de Java*: Dado que es una aplicación Java, necesitamos tener instalado el entorno de ejecución Java Runtime Environment (JRE).

CAPÍTULO 4
Conclusiones

Como se ha indicado a lo largo del proyecto la aplicación *d*ANIMA busca facilitar el trabajo a los desarrolladores de programas Maude. Para ello se ha realizado una aplicación que ayuda a visualizar y comprender la ejecución y facilita toda la comunicación con el intérprete Mau-Dev.

Los objetivos alcanzados han sido los siguientes:

- Se ha proporcionado un editor de código Maude con todas las características que este componente debe aportar.
- Se ha implementado un componente que representa y explora los estados del árbol.
- Proponemos una forma distinta de comunicación con el intérprete que soporte los casos susceptibles de bloquear el navegador.
- Ofrecemos una alternativa a los desarrolladores que no tengan acceso a la red, a la vez que les permite aprovechar mejor todos los recursos del sistema.

Para concluir, nos gustaría destacar que *d*ANIMA proporciona un entorno de análisis y depuración versátil y potente que contribuye a promover el empleo del lenguaje Maude y de los métodos formales en la industria del Software.

Bibliografía

[1] J.Messeger, *Rewriting as a unified model of concurrency*. In Proceedings of the Concur'90 Conference. 1990: 384-400.

[2] Matthias Felleisen, Philippa Gardner, *Programming Languages and Systems*. 22nd European Symposium on Programming, 2013: 122-123.

[3] *RichTextFX*. Disponible en https://github.com/TomasMikula/RichTextFX.

[4] *Anima Online Stepper*. Disponible en http://safe-tools.dsic.upv.es/anima/.

[5] María Alpuente, Demis Ballis, Francisco Frechina y Julia Sapiña Journal of Symbolic Computation. *Exploring Conditional Rewriting Logic Computations*, 69:3-39, Julio-Agosto, 2015.

[7] John Q. Walker II. Software: Practice and Experience. *A node-positioning algorithm for general trees*, 20(7):685–705, 1990.

[8] *Apache Maven Proyect*. Disponible en https://maven.apache.org/.

[9] Santiago Escobar. *Functional Logic Programming in Maude.*. DSIC-ELP, Universitat Politècnica de València, Spain.

[10] Manuel Clavel, Francisco Durán, Steven Eker, Santiago Escobar, Patrick Lincoln, Narciso Martí-Oliet, J. Meseguer, Carolyn Talcott. *Maude Manual.*. Marzo 2015.

[11] *Maude-NPA*. Disponible en http://maude.cs.uiuc.edu/tools/Maude-NPA/.

[12] *Mau-Dev*. Disponible en http://safe-tools.dsic.upv.es/maudev/.

[13] Joseph A. Goguen, Claude Kirchner,Hélène Kirchner, Aristide Mégrelis, José Meseguer, Timothy C. Winkler. An Introduction to OBJ 3. *Conditional Term Rewriting Systems*. 1st International Workshop Orsay, France, July 8-10, Proceedings, 1987:258-263.

[14] *Oracle Technology Network for Java Developers*. Disponible en http://www.oracle.com/technetwork/java/index.html.

APÉNDICE A
Manual de Usuario

Índice general

Capítulo 1

Introducción

El proyecto *d*ANIMA es una aplicación informática dirigida a facilitar el trabajo a desarrolladores de aplicaciones en el lenguaje Maude. Esta herramienta proporciona a los usuarios finales las siguientes ventajas:

- Facilita la visualización de código fuente en Maude mediante un editor de código incluido en el entorno de desarrollo integrado, el cuál resalta la sintaxis propia de este lenguaje de programación.
- Facilita la depuración de un programa Maude dada la representación o forma de árbol del .

Este documento le permitirá aprender a utilizar todas las funcionalidades básicas que componen la aplicación ANIMA.

.

Capítulo 2

Pantalla Inicial

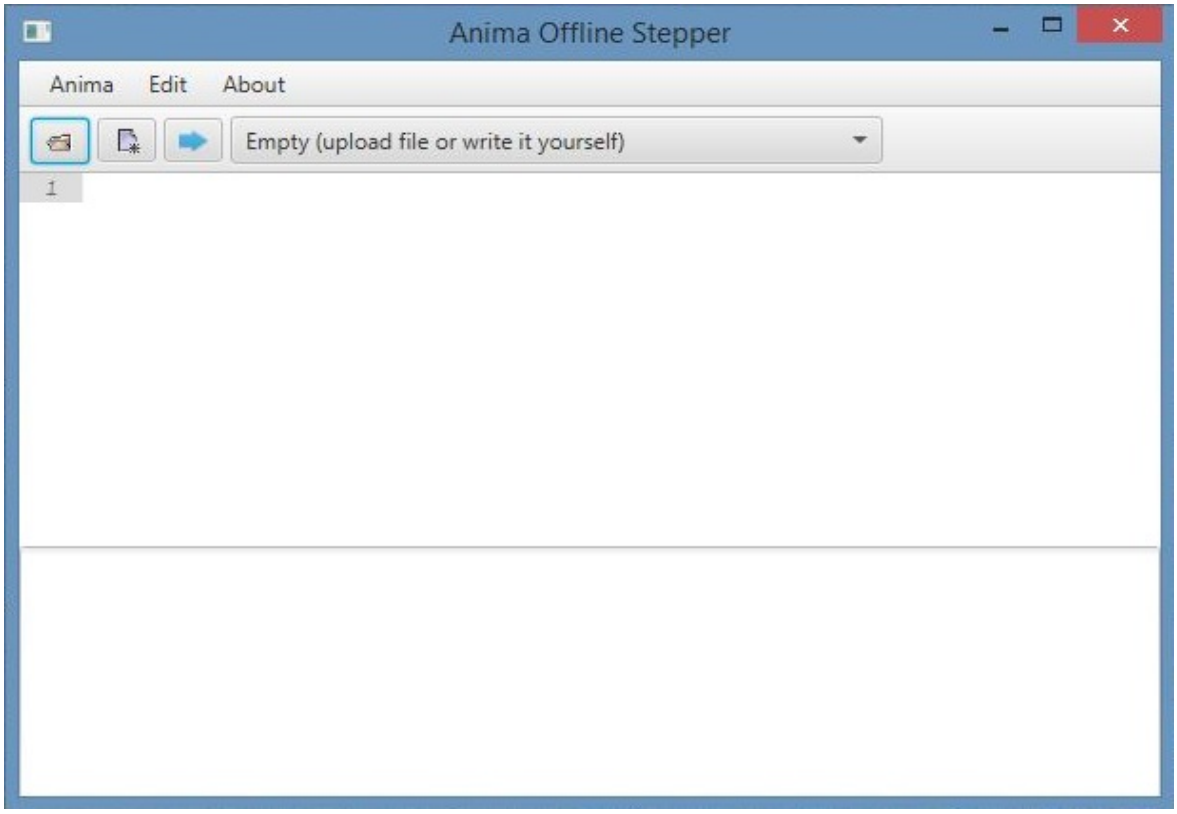

Figura 2.1: Pantalla Inicial de la aplicación

Tras abrir la aplicación visualizaremos esta pantalla, la pantalla inicial de la aplicación, donde tenemos varias opciones a elegir. Podemos escribir códi-

go directamente en el área central de la ventana o bien elegir algún programa ya desarrollado para ser cargado en el sistema.

Para elegir un programa deberá pulsar sobre la pestaña desplegable de arriba, la cual tiene escrito: “Empty (upload file or write it yourself)”

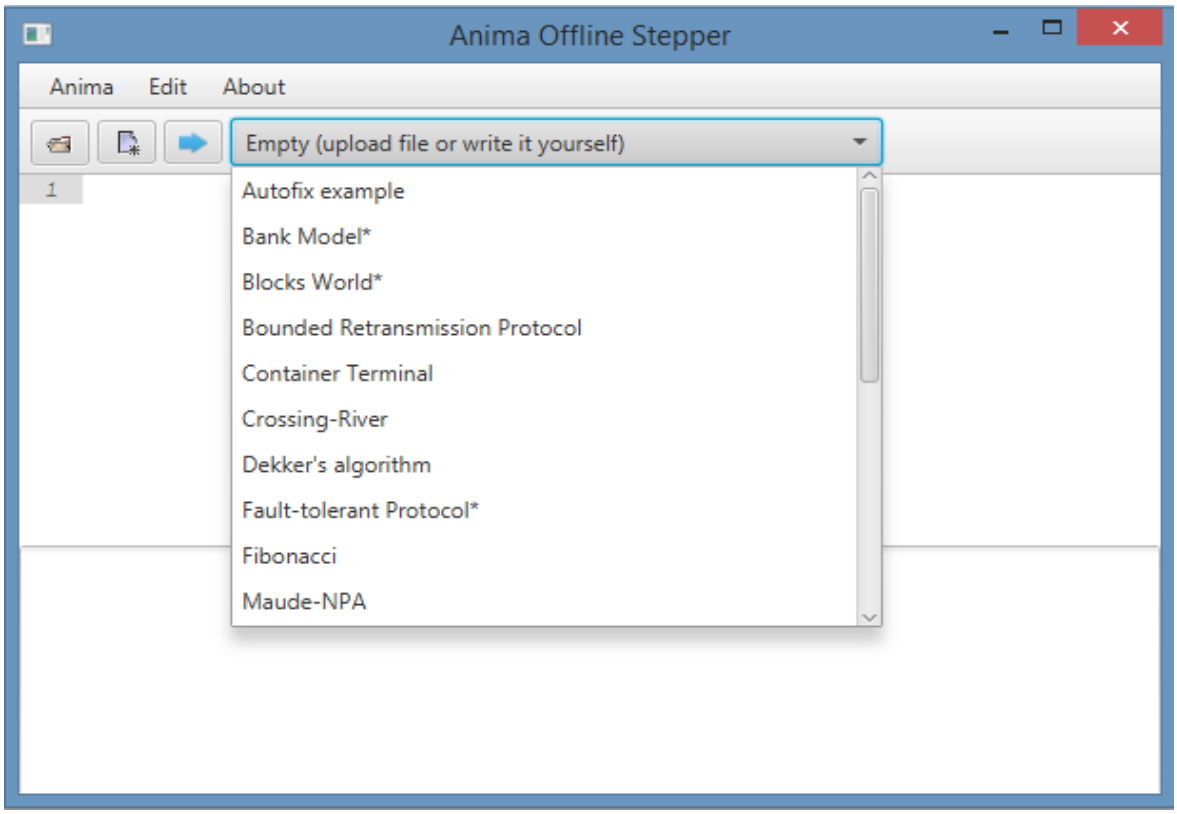

Figura 2.2: Pantalla Inicial con la pestaña de códigos disponibles

Observaremos una lista con los posibles códigos a utilizar. Para movernos por la pestaña tenemos unas flechas y una barra que podemos mover hacia la posición de la lista que queramos visualizar.

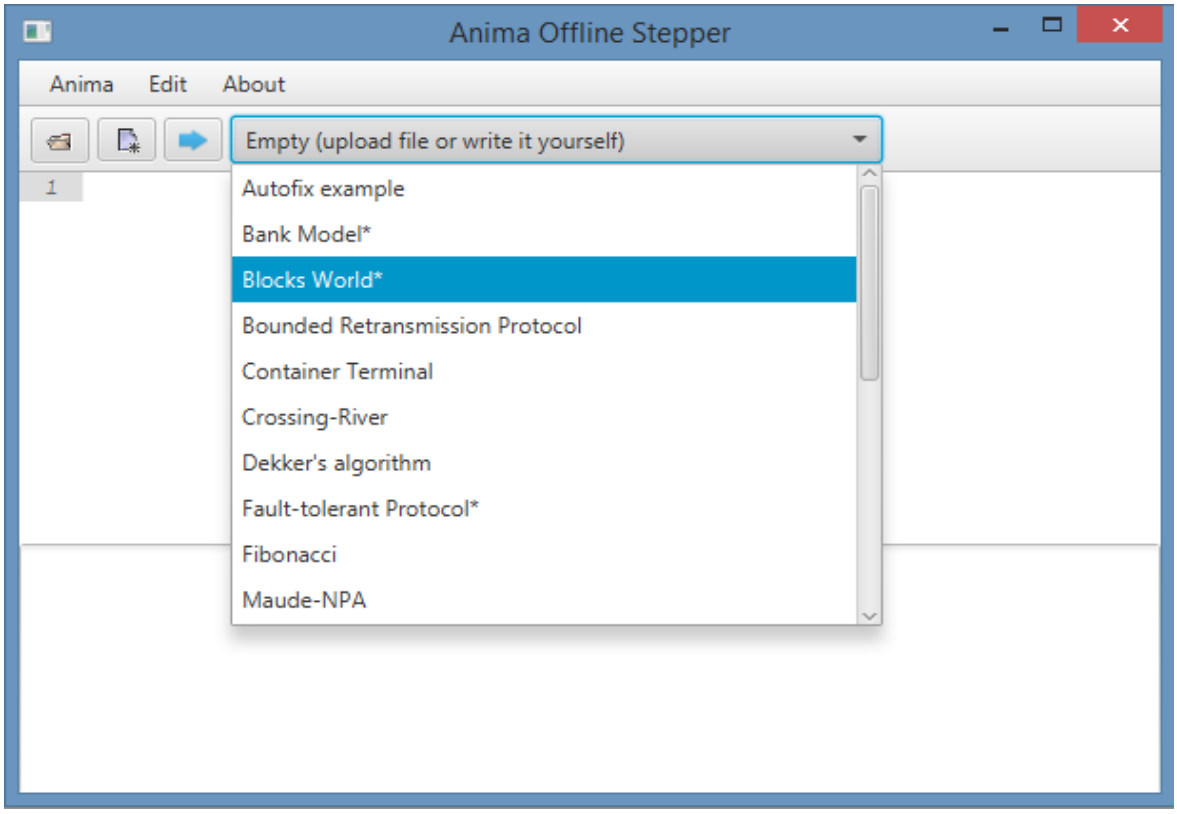

Figura 2.3: Pantalla Inicial con la pestaña de códigos disponibles y un código seleccionado

Tras esto podemos volver a pulsar sobre la pestaña para cerrarla o bien seleccionar algún código. Cuando pasemos el ratón por la lista se indicará cuál es el código seleccionado.

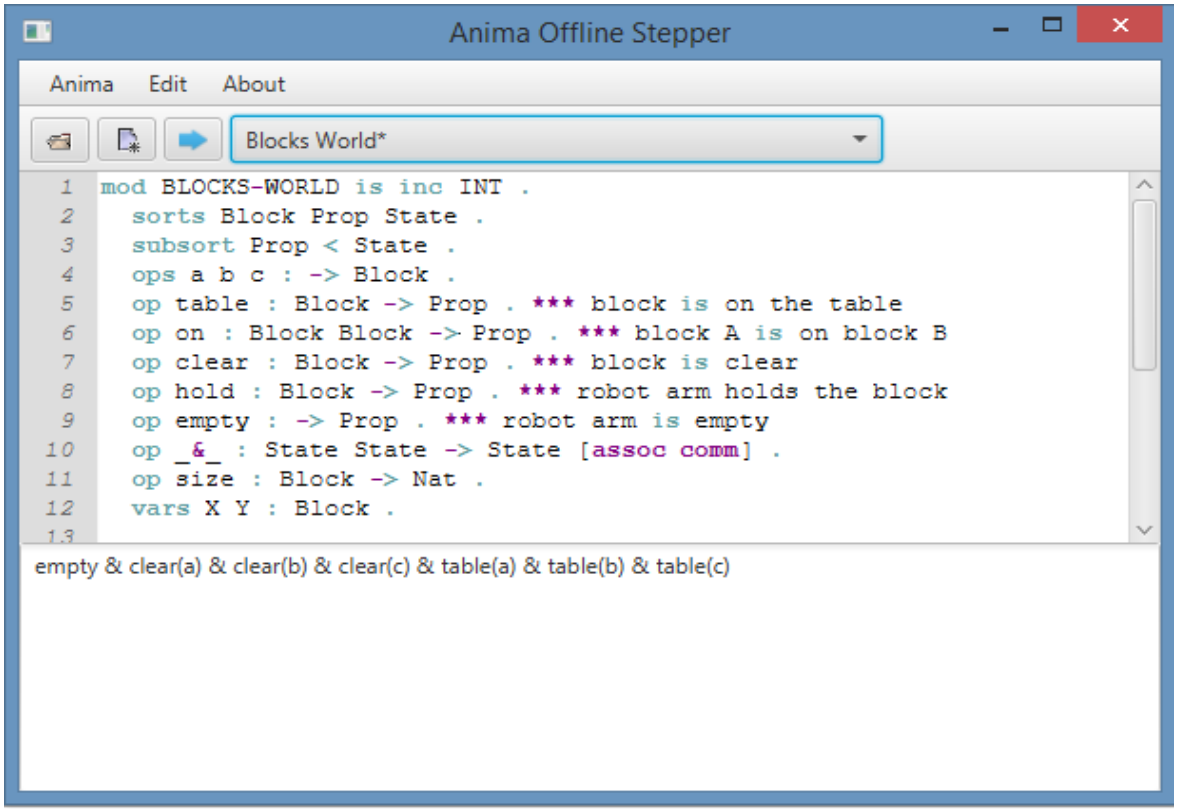

Figura 2.4: Pantalla Inicial con un código seleccionado

Tras pulsar sobre el código seleccionado, se cerrará la pestaña con la lista de programas y se rellenarán dos área de texto, una con el código propiamente dicho y otra con el estado inicial asociado.

Podremos ver trozos de código resaltado debido a que son palabras clave del lenguaje utilizado. También se pueden editar tanto el código como el estado.

Tras hacer todos los cambios pertinentes, es necesario pulsar el botón con una flecha azul para comenzar con la depuración del programa.

Capítulo 3

Pantalla de Depuración

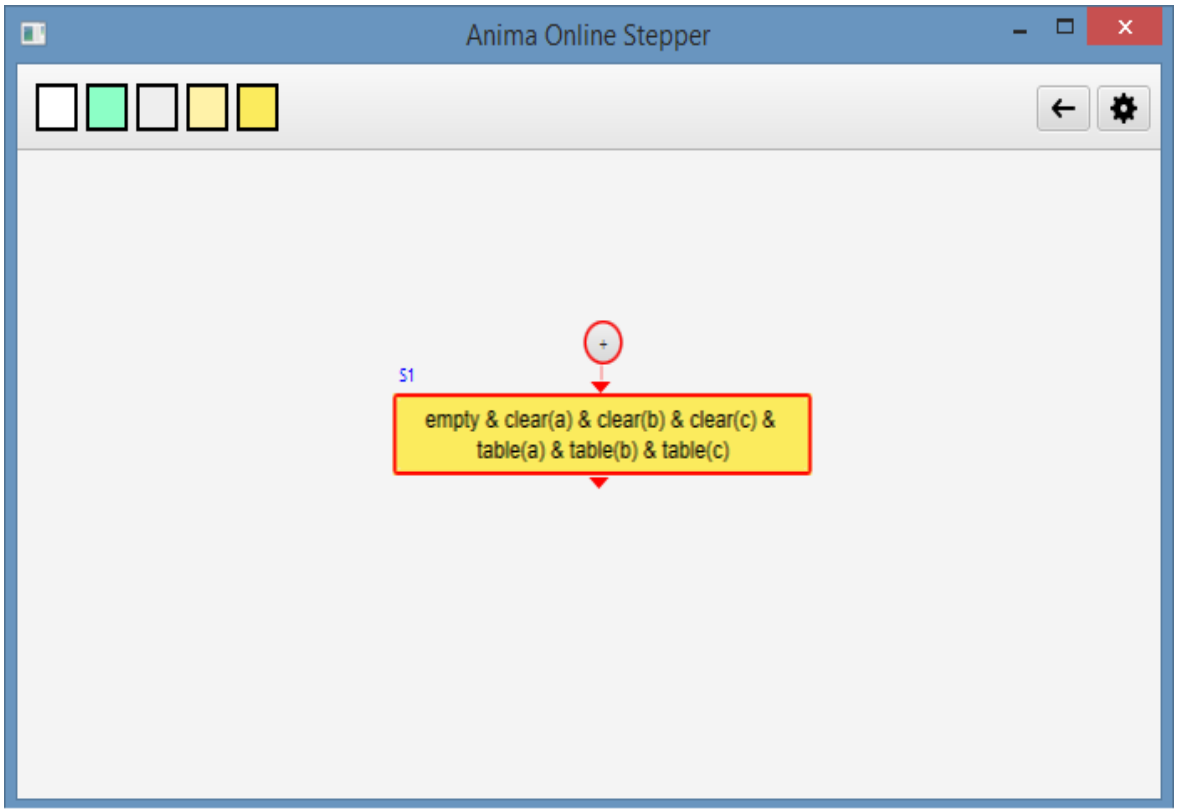

Figura 3.1: Pantalla Depuración del código seleccionado

La primera pantalla para depurar un programa tendrá un aspecto similar a este, donde podemos ver que es el nodo 1 en el árbol, teniendo un nodo

anterior no normalizado, para visualizarlo deberemos pulsar sobre el botón que tiene el signo de suma.

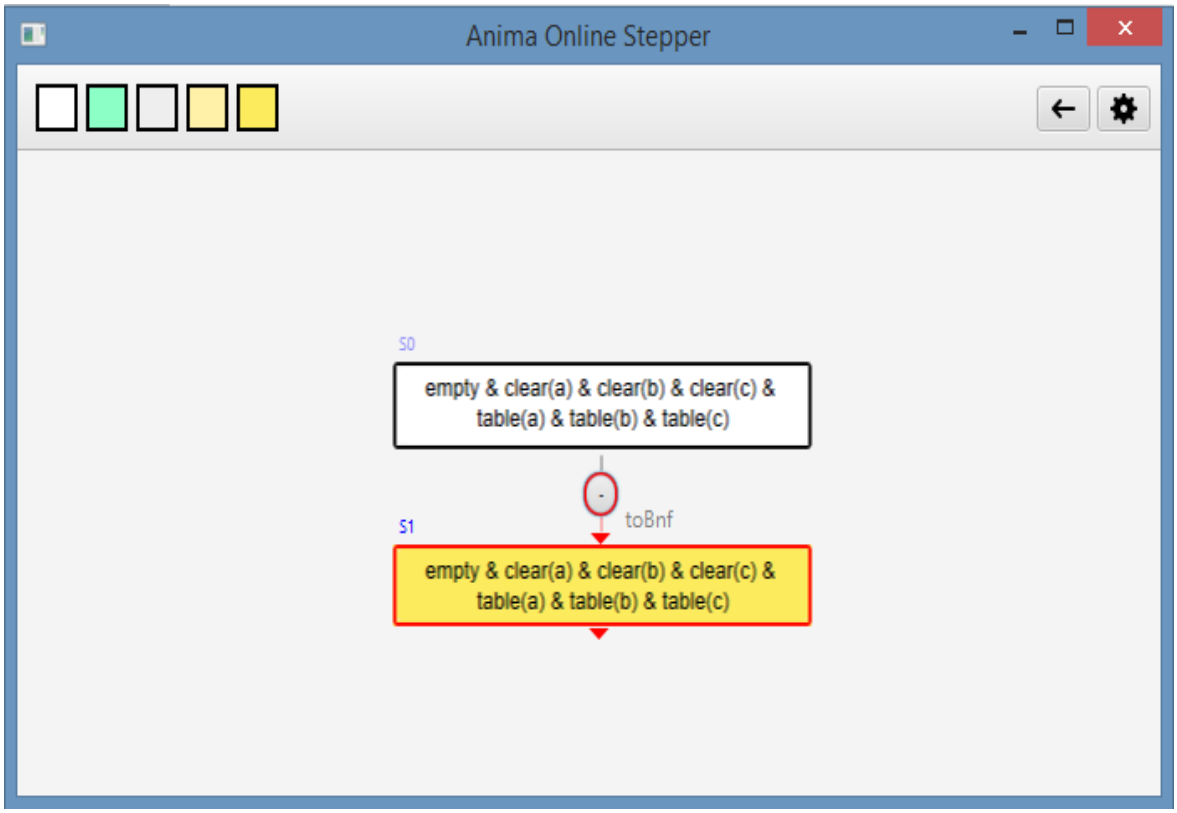

Figura 3.2: Nodo 1 desplegado

Tendremos el nodo inicial del programa, donde su versión normalizada es el nodo 1. Para volver a plegar el nodo 1 deberemos pulsar sobre el botón con el signo de resta.

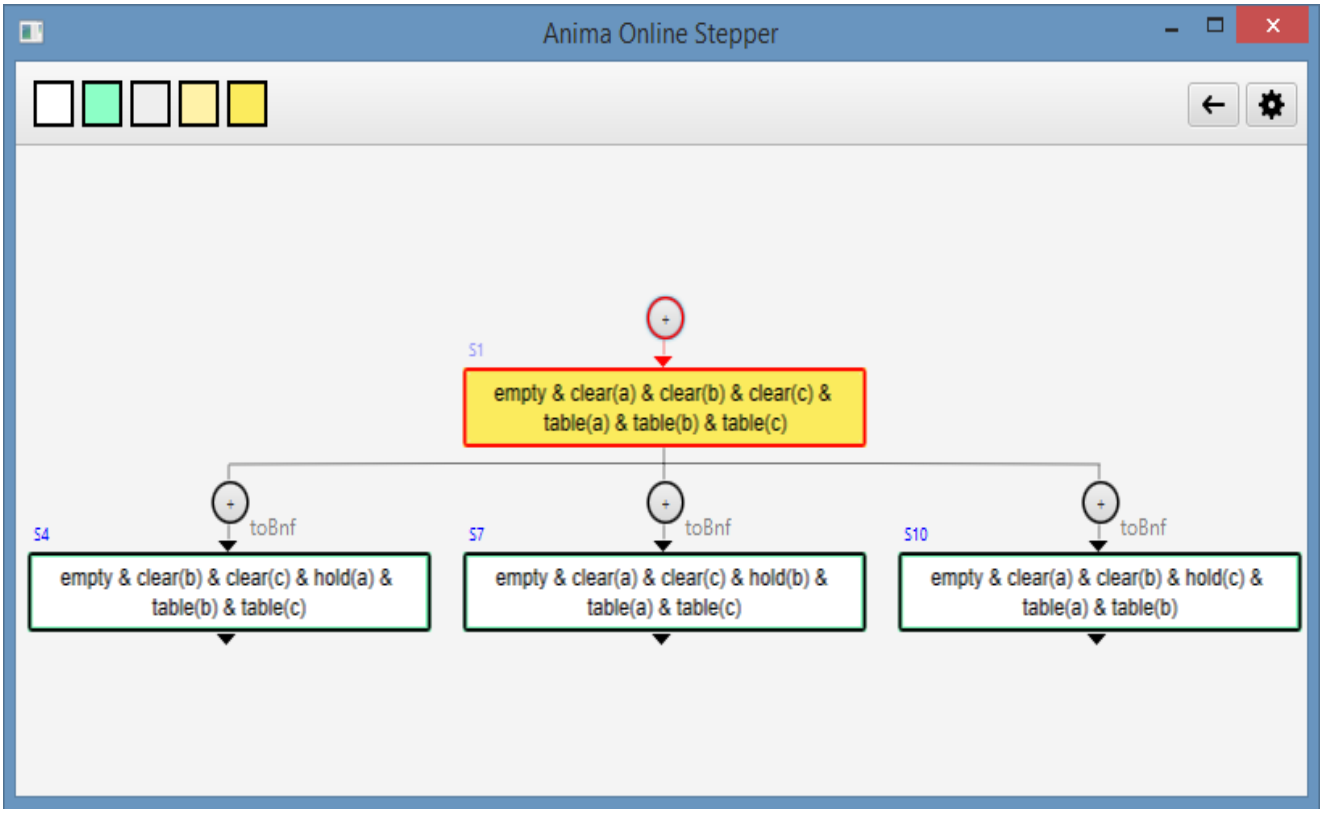

Figura 3.3: Nodo 1 expandido

Y finalmente, podemos pulsar sobre el botón donde pondrá una S y el número de nodo en el árbol, expandiendo los nodos que dependen de dicho estado. Otra forma de expandir dicho nodo podría ser mediante el menú contextual de la aplicación, pulsando el botón derecho del ratón donde visualizaremos varias opciones entre ellas expandir el nodo.

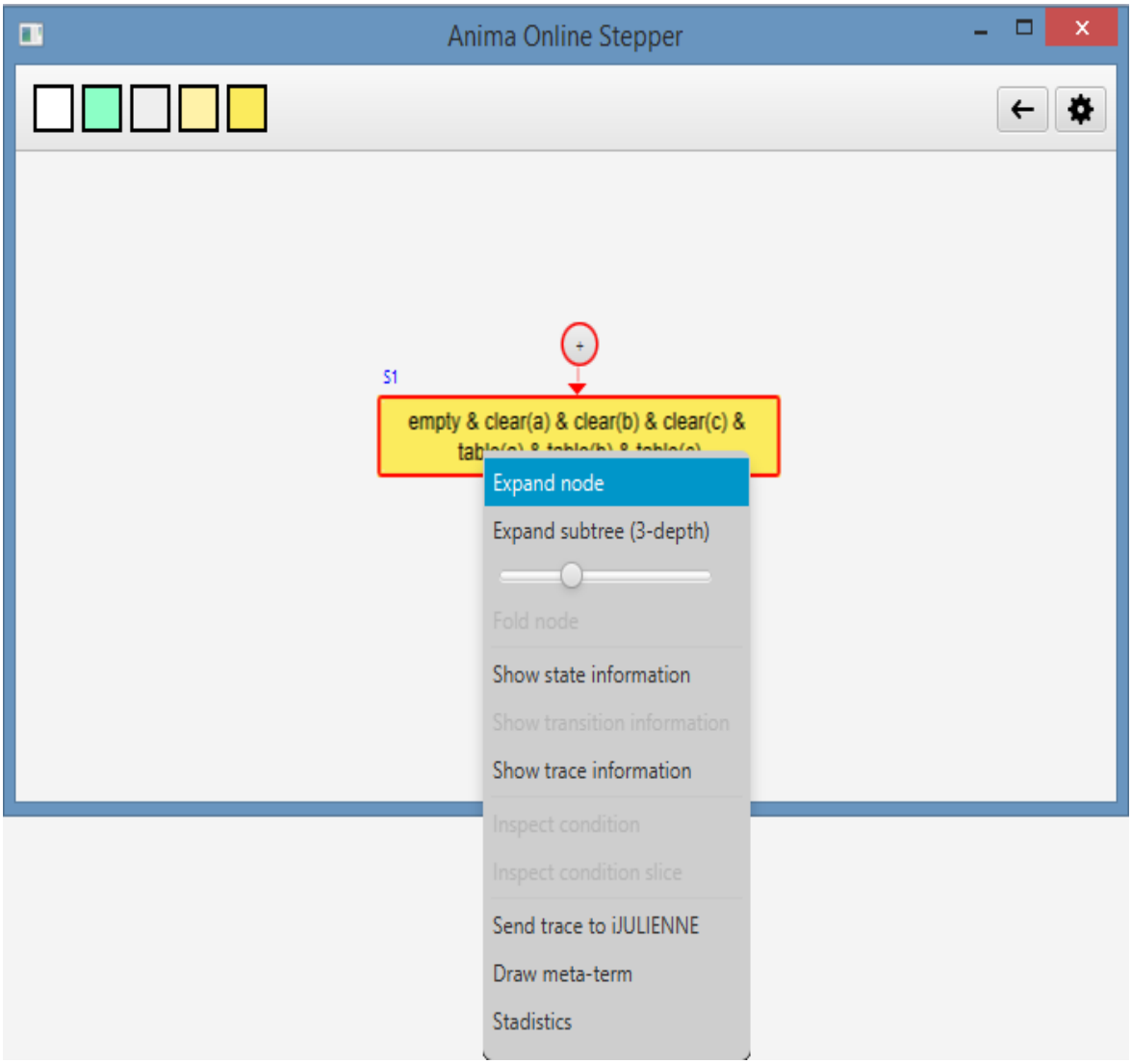

Figura 3.4: Menu Contextual de la aplicación