Interfaz gráfica para Mobile Maude

Proyecto de Sistemas Informáticos,
Facultad de Informática,
Universidad Complutense de Madrid

Autores

Irene González Palomar
Diego González Rodríguez
Luis Julián Plaza Fernández

Profesor director: Alberto Verdejo

Curso académico: 2006- 2007

1. RESUMEN DEL PROYECTO

1.1. TITULO Interfaz gráfica para Mobile Maude

1.1.1. RESUMEN

La finalidad de este proyecto es desarrollar un conjunto de aplicaciones para poder observar gráficamente el flujo de ejecución de aplicaciones Mobile Maude, pudiendo ver los objetos, los estados por los que van pasando dichos objetos en los distintos terminales de ejecución, las reglas que se aplican y los mensajes que se envían.

Este proyecto servirá para tres tipos de uso:

- Una versión distribuida que consiste en ejecutar varias veces la aplicación y en cada una de ellas se cargará el programa a ejecutar y una instancia de éste. Estas instancias interaccionarán entre sí provocando que los objetos móviles viajen y, creando los estados y mensajes que mostraremos en la aplicación correspondiente junto con los objetos.
- Otra versión centralizada que permitirá realizar exactamente lo mismo que en la distribuida pero sólo ejecutando una vez la aplicación y cargando el programa a ejecutar. En la misma interfaz se mostrarán las distintas instancias y los distintos estados, objetos y mensajes que se crean en cada instancia al viajar los objetos móviles.
- Y una última versión que chequea modelos e indica si es cierto o falso. En el caso de que sea falso mostrará un contraejemplo.

1.2. PROJECT *TITLE* Graphical interface for Mobile Maude

1.2.1. SUMMARY

The purpose of this project is to develop a set of applications to be able to observe graphically the flow of execution of Mobile-Maude applications, being able to see the objects, the states through which they are passing in the different terminals of execution, the rules that are being applied and the messages that are sent.

This project will be used for three types of versions:

- A distributed version that consists of executing several times the application and each one of them will load the program to execute and an instance of this one. These instances will interact to each other causing that the mobile objects travel and creating the states and messages that we will along with show in the corresponding application the objects.
- Another centralized version that will allow making exactly just like in the distributed one but only executing the application once and loading the program to execute. In the same interface we will see the different instances and the different states, objects and messages that are created in each instance when traveling the mobile objects.
- And a last version that checks models and indicates if it is certain or false. In case it is false shows a counterexample.

2. INTRODUCCIÓN

A la hora de elegir nuestro proyecto de Fin de Carrera buscábamos algo que se adaptara a nuestros gustos personales y a la vez que nos sirviera como experiencia para nuestra futura vida laboral.

A los tres participantes de este proyecto una de las cosas que más nos atrae y que nos empujó a estudiar Informática es el desarrollo de proyectos y la programación. Por ello al conocer el tema enseguida nos interesó y nos pusimos en contacto con el Profesor que lo dirigiría para que nos explicara con más detalle en que consistía.

Básicamente una de las razones que nos han llevado a interesarnos en el tema es que consiste en desarrollar una Interfaz Gráfica en JAVA, lenguaje que ya conocíamos bastante antes de empezarlo. Esto nos va a permitir coger más soltura a la hora de programar y para tener más experiencia desarrollando proyectos en este lenguaje. Además nos puede aportar no sólo experiencia a la hora de desarrollar un proyecto en el terreno laboral sino que nos va a servir para saber solucionar problemas que nos vayan surgiendo, aprender a desenvolvernos nosotros mismos, ponernos objetivos y lograrlos en un tiempo determinado. De hecho el desarrollar este proyecto y cualquier otro conlleva varias tareas que se pueden parecer a las que se desarrollan en una empresa, tales como dirigir, planificar el desarrollo, la implementación, las pruebas…

A parte de sus beneficios nos va a servir para conocer ciertos lenguajes (Maude, Mobile Maude) que antes desconocíamos y que nos pueden servir en un futuro, así como para tener más experiencia en la plataforma Linux, ya que aunque en la carrera la hemos usado, no tenemos la misma soltura que con otros sistemas operativos y siempre va a venir bien saber desenvolverse en distintos medios.

Por lo tanto con todo esto esperamos que con él se cumplan tanto nuestras expectativas, como se llegue a alcanzar el objetivo que se ha marcado nuestro Tutor con este proyecto.

3. OBJETIVOS

3.1. Finalidad del proyecto.

La finalidad de este proyecto es desarrollar un conjunto de aplicaciones en JAVA que interaccionan con Maude y Mobile Maude que permitirá observar gráficamente el flujo de ejecución de aplicaciones Mobile Maude, pudiendo ver los estados por los que van pasando los objetos en los distintos terminales de ejecución, las reglas que se aplican y los mensajes que se envían.

Cuando ejecutamos una instancia de una aplicación Mobile Maude en un terminal se muestra la traza de la ejecución. Este proyecto básicamente consiste en analizar la traza de la ejecución de cada instancia del programa a ejecutar y mostrarla gráficamente mediante una interfaz.

Por lo tanto la aplicación está compuesta básicamente por una interfaz gráfica sencilla mediante la cual se cargan los programas y sus instancias, y se muestran los atributos más característicos y los estados a través de los cuales pasan los objetos. Se mostrarán aquellos estados a los que se llega tras aplicar aquellas reglas que se han definido como importantes y los mensajes que se envían también así definidos.

3.2. Ventajas del proyecto.

Las ventajas que se quieren obtener con este proyecto son:

- Programa multiplataforma.
- Facilitar el análisis de la traza de la ejecución de un programa de Mobile Maude a través de la conexión entre JAVA y Maude.
- Visualizar gráficamente la traza de la ejecución de manera más sencilla pero equivalente a la realizada en un terminal.
- Elegir las reglas que se quieren mostrar.
- Mostrar sólo aquellos estados que se consiguen con las reglas elegidas.
- Mostrar los mensajes que se envían entre los Objetos Móviles.
- Poder navegar por los estados que se hayan creado a gusto del usuario.

- Poder elegir entre la versión distribuida con varias interfaces como la versión centralizada, con una sola interfaz.
- Visualizar la traza correspondiente a cada estado.
- Interfaz sencilla de usar.

3.3. Medios Materiales

- Java JDK 1.5
- Eclipse 3.2
- Maude versión 2.3
- Mobile Maude
- PC con Sistema operativo Linux.
- Microsoft Word 2003

4. MAUDE

Maude es un lenguaje basado en ecuaciones, para describir la parte estática de un sistema, y reglas, para describir la parte dinámica o transiciones de un sistema. El sistema Maude permite especificar, programar y razonar sobre lenguajes de programación, modelos de cómputo y sistemas software en general.

Es un metalenguaje de alto rendimiento y un sistema que soporta el cómputo tanto de especificaciones ecuacionales como de lógica de reescritura, para una amplia gama de aplicaciones. Maude ha sido desarrollado con la idea de ser usado como lenguaje de programación de alto nivel para diversas aplicaciones. A pesar de tratarse de un lenguaje formal, destaca por su alto grado de eficiencia, lo que permite pensar en Maude como un candidato real para el desarrollo de aplicaciones. Por tratarse de un lenguaje que permite especificación ecuacional, es especialmente adecuado para el desarrollo de aplicaciones del ámbito de los metalenguajes. En este sentido ha sido usado con bastante éxito para el manejo de protocolos, solución a problemas de planificación y tratamiento, ejecución de diversas lógicas, demostradores de teoremas y modelos de cómputo. Sin embargo, su interfaz es bastante primaria, lo que impide que éste pueda llegar a un mayor número de usuarios y pueda ser usado más allá del ámbito puramente académico o investigador.

Maude contiene un sublenguaje funcional basado en la lógica ecuacional de pertenencia que extiende el algebra con tipos ordenados de OBJ3, y permite la definición de tipos, subtipos, operaciones sobrecargadas y operaciones parciales. La lógica ecuacional de pertenencia se puede considerar una sublógica de la lógica de reescritura, pero sin embargo las reglas de reescritura no admiten una interpretación ecuacional, pues pueden no terminar ni ser confluentes. Por esta razón, en general a partir de un estado inicial hay muchos cómputos diferentes, por lo que resulta crucial usar estrategias adecuadas que controlen la ejecución de cada módulo. En Maude, tales estrategias no caen fuera del marco lógico, sino que son estrategias internas, definidas mediante reglas de reescritura al metanivel. Esto es posible porque la lógica de reescritura es reflexiva.

Maude soporta de manera eficiente la reflexión a través de un modulo META-LEVEL, que permite no sólo la definición y ejecución de estrategias internas mediante reglas de reescritura, sino también muchas otras aplicaciones, incluyendo la metaprogramación y un algebra de módulos extensible.

Las características destacables de Maude son las siguientes:

- **Potente:** Puede modelar casi todo, desde el conjunto de los números naturales pasando por un sistema biológico hasta programar el lenguaje Maude a sí mismo. Cualquier cosa que se pueda escribir, hablar o describir mediante el lenguaje humano, se puede expresar con instrucciones Maude.

- **Simple:** Su semántica está basada en los fundamentos de la teoría de categorías, la cual es bastante intuitiva y directa, hasta que un matemático empezó a describirla formalmente con símbolos y letras griegas. Comparado con la mayoría de los lenguajes, Maude no tiene mucha sintaxis que memorizar, solo un pequeño conjunto de palabras claves y símbolos, y algunas directivas generales y convenciones a seguir.

- **Flexible y expresivo,** capaz de representar cambios en sistemas con estructuras muy diferentes. Permite sintaxis definida por el usuario, con una completa libertad a la hora de elegir los operadores y las propiedades estructurales apropiadas para cada problema.

- **Desafiante**. Puede llegar a solucionar lo más difícil o complejo. Esto desafía al programador a ser astuto, en vez de resolver el problema afrontándolo con una serie de variables globales y funciones que a menudo son un caos de por sí.

- **Concurrente.** Soporta el modelado de sistemas concurrentes orientados a objetos, de una forma simple y directa.

Extensiones de Maude:

- **Core Maude** es el intérprete de Maude 2.3 implementado en C++ y que provee la funcionalidad básica del lenguaje.

- **Full Maude** es una extensión de Maude escrita en el mismo Maude que soporta un avanzado sistema de módulos parametrizados, teorías, vistas, y expresiones de módulos al estilo de OBJ, con sus correspondientes operaciones de instanciación, composición y aplanamiento. Como caso especial de dichos módulos, Full Maude proporciona soporte sintáctico para la especificación y programación de sistemas concurrentes orientados a objetos. Otra aplicación destacable es un demostrador inductivo de teoremas, conocido como ITP (Inductive Theorem Prover), que es un asistente para demostrar propiedades inductivas de programas funcionales e implementa un sistema formal que incluye, entre otras, reglas de inferencia para razonar por inducción.

Así pues, Maude es un lenguaje declarativo de alto nivel que permite especificar sistemas computacionales basados en la lógica de reescritura para una amplia variedad de aplicaciones, incluyendo el desarrollo de herramientas para la demostración de teoremas y para el prototipado de lenguajes, la especificación y análisis de sistemas concurrentes y distribuidos, así como la representación y ejecución de diversas lógicas.

4.1. CORE MAUDE

En Maude los elementos sintácticos básicos son los identificadores, usados para poner nombre a módulos, tipos y operadores. En Maude un identificador es una cadena ASCII. Algunos caracteres como ‘{‘ y ‘[‘ son identificadores válidos, o pueden formar parte de identificadores multitoken, pero tienen la particularidad de que rompen los tokens. El carácter ‘‘’ es usado como un carácter de escape indicando un espacio o que un carácter especial no rompa la secuencia. Por ejemplo la secuencia 1‘ab‘{c‘,d‘}ef es solamente un identificador . Maude lo mostraría como un identificador de la forma ‘1 ab{c,d}ef’.

Las unidades básicas de especificación y programación son los módulos. En Core Maude hay dos tipos de módulos: los módulos funcionales y los módulos de sistema.

Lo primero que una especificación necesita es declarar los tipos (denominados sorts) de los datos definidos y las correspondientes operaciones. Los sorts están parcialmente relacionados mediante relaciones de subtipo (subsort) definiéndose un orden parcial.

La lógica en la que se basa Maude es la lógica ecuacional de pertenencia. En esta lógica los tipos se agrupan en clases de equivalencia llamados kinds. Para este propósito, dos tipos se consideran equivalentes si pertenecen al mismo componente conexo. En Maude los tipos están definidos por el usuario mientras que los “Kinds” son implícitamente asociados con componentes conexos de tipos y son considerados como “supertipos de error”.

Las variables en Maude se declaran restringiéndose a un determinado dominio de un tipo o “kind”.

Un término es una constante, una variable, o la aplicación de un operador a una lista de términos argumento. El tipo de una constante o variable es su sort declarado. En la aplicación de un operador, la lista de argumentos tiene que coincidir con la aridad declarada del operador.

Además, los módulos pueden tener ecuaciones y axiomas de pertenencia y reglas, esto será explicado más adelante cuando se expliquen los módulos funcionales y de sistema.

4.2. FULL MAUDE

Durante el desarrollo del sistema Maude se ha puesto especial énfasis en la creación de facilidades de metaprogramación para permitir la generación de entornos de ejecución para un amplio número de lenguajes y lógicas. El área más obvia donde Maude puede ser usado como un metalenguaje es a la hora de construir extensiones para sí mismo.

Ha sido posible definir en Core Maude una extensión para el lenguaje con una notación para programación orientada a objetos, módulos parametrizados, vistas, y expresiones de módulos. Gracias a la eficiente implementación del motor de reescritura, el analizador léxico, y el módulo META-LEVEL, las extensiones del lenguaje se ejecutan con una eficiencia razonable. En el futuro, es posible que mucha de esta funcionalidad soportada en Full Maude se soporte en Core Maude. Full Maude contiene Core Maude como un sublenguaje, de manera que los módulos Core Maude también pueden ser introducidos en Full Maude.

Dado que el entorno de ejecución para Full Maude ha sido implementado en Core Maude, para inicializar el sistema y empezar a usarlo, la primera cosa que tenemos que hacer es cargar el módulo FULL-MAUDE en el sistema. Para iniciarlo sólo hay que ejecutar el comando in o load.

Para inicializar el sistema se usa el comando loop init. Este comando arranca un input-eval-loop para permitir la interacción con el usuario mediante la introducción de módulos, teorías, vistas, comandos y mantener una base de datos en la cual almacenar todo lo introducido. En el caso de que se pulse Ctrl + C puede ocurrir que se rompa el loop, y con él la ejecución actual de Full Maude. Maude puede intentar recuperar el loop por si mismo, pero si esto no funciona podemos reiniciarlo de nuevo mediante el comando loop.

Todo lo introducido en Full Maude tiene que introducirse entre paréntesis. Dado que Core Maude sigue activo, este manejará cualquier comando o módulo introducido que no esté entre paréntesis. Esto permite la posibilidad de usar ambos sistemas al mismo tiempo. Gracias a esto podemos usar comandos Core Maude cuando estemos usando Full Maude. Esto puede llevar a cierta confusión siendo necesario tener cuidado de poner o no poner paréntesis según corresponda. Full Maude y mantiene una base de

datos independiente de la que usa Core Maude para almacenar los módulos introducidos en el sistema.

4.3. La Traza en Maude

Las facilidades de traza nos permiten seguir la ejecución de nuestras especificaciones, es decir, las secuencia de reescrituras o simplificación de reducciones tienen lugar. La traza se activa con el comando:

set trace on .

Como la traza es a menudo voluminosa, existe cierta cantidad de opciones para su control. Una de las más útiles es la traza selectiva:

set trace select on .

trace select foo bar ([_,_]) .

Esto provocará que se visualice solamente las reescrituras donde la orden está etiquetada con el nombre seleccionado (en el ejemplo anterior: foo, bar ([_,_])) ó la reducción esté encabezada con operadores etiquetados con el nombre que se encuentra seleccionado para trazar.

Una opción útil para la metaprogramación es:

trace exclude FOO BAR .

Esta opción excluirá de la traza a los módulos nombrados por FOO ó BAR. Gracias a la opción trace exclude, podemos excluir de la traza módulos del metanivel.

Para poder ver paso a paso la traza, pudiendo ver así el cambio que se produce en los atributos de los objetos y como viajan según las reglas que se van aplicando usaremos la opción

cont 1

Para ocultar información sobre la aplicación de las ecuaciones se usa el comando

set trace eq off .

En la ejecución de la traza aparecen varias reglas. Si sólo nos interesa alguna o varias reglas en particular podemos usar la siguiente instrucción, la cual sólo nos sacará la información referente a la aplicación de la reglas/s que hemos indicado

trace select rule

donde rule es la regla o reglas que queremos mostrar.

Por otro lado si queremos sacar la misma traza pero sin mostrar las substituciones que se hacen usaremos el comando

set trace substitution off .

Otra opción para reducir la traza es omitir la evaluación de las condiciones de la siguiente manera

set trace condition off .

Como observamos con los comandos que hemos explicado anteriormente y algunos otros, se tiene la posibilidad de adaptar la traza a nuestras necesidades mostrando u ocultando la información que se crea necesaria analizar.

5. MOBILE MAUDE

Mobile Maude es un lenguaje de agentes móviles declarativo, simple y general que extiende al lenguaje Maude para soportar cómputos móviles. Mobile Maude está especificado formalmente en Maude. Ya que esta especificación es ejecutable, puede utilizarse como prototipo del lenguaje, en el cual se pueden simular sistemas de agentes móviles.

La última versión de Maude soporta la comunicación con objetos externos e incluye una implementación de sockets como el primero de tal clase de objetos. Esta disponibilidad de comunicación entre diferentes procesos Maude ha permitido implementar Mobile Maude para que sea realmente un lenguaje distribuido. Una parte de la especificación de Mobile Maude se encarga de redirigir los mensajes entre objetos móviles a través de sockets cuando el emisor y el receptor no se encuentran en el mismo proceso. Los objetos móviles, junto con su código, también son enviados a través de sockets cuando quieren viajar de un proceso a otro.

Las principales características del lenguaje Mobile Maude son las siguientes:

- Basado en la lógica de reescritura, que facilita la especificación de sistemas concurrentes, distribuidos y abiertos, incluyendo datos, eventos, la evolución del sistema, las propiedades de seguridad, y aspectos de tiempo real.

- Orientado a objetos, con soporte de primer nivel para objetos y agentes distribuidos y su comunicación, tanto síncrona como asíncrona, sin perderse en modo alguno su semántica formal.

- Reflexivo, con soporte formal para la metaprogramación a cualquier nivel, y para la reconfiguración dinámica con una gran flexibilidad.

- De amplio espectro, permitiendo la transformación y refinamiento riguroso partiendo de especificaciones para llegar a programas declarativos eficientes.

- Modular, con un avanzado sistema de módulos que soportan parametrización, teorías para requisitos, vistas, y expresiones de módulos, con sus correspondientes operaciones de instanciación, composición y aplanamiento.

- Móvil, con soporte para el movimiento de datos, estados locales, y programas, y facilitando la integración de código escrito en lenguajes convencionales, como C.

- Con una base formal para el desarrollo de modelos de seguridad y la verificación de propiedades sobre tales modelos.

Todas ellas lo convierten en el lenguaje adecuado para realizar la especificación y prototipado de sistemas basados en agentes móviles, consiguiéndose los objetivos indicados más arriba.

Los dos elementos básicos de Mobile Maude son los procesos y los objetos móviles o agentes. Los procesos son ambientes computacionales situados en una cierta localidad en los que los agentes residirán. Cada agente dispone de su propio estado interno y de su código; se pueden mover de un proceso a otro, bien entre diferentes procesos en la misma localidad o entre procesos en diferentes localidades; y se pueden comunicar entre sí de modo asíncrono, mediante intercambio de mensajes, independientemente de su localización.

La clave para la especificación de los agentes reside una vez más en la capacidad reflexiva de la lógica de reescritura, que permite representar el código dentro de un agente como un dato más. De esta forma el código interno de un agente no está sujeto a ninguna restricción pudiendo tratarse de un programa cualquiera en Maude. Por otra parte, la semántica de los aspectos noveles relacionados con la movilidad se obtiene de forma considerablemente simple, mediante un conjunto de reglas general e independiente de aplicaciones concretas.

Los tipos de objetos que nos podemos encontrar pueden ser objetos móviles o objetos Servidor o Cliente.

Los objetos móviles están especificados por la clase MobileObject. Puede haber uno o más de uno y no son fijos ya que viajan y pueden aparecer o desaparecer. La clase MobileObject tiene la siguiente estructura:

```
Class MobileObject |
        mod : Module,
        s: Term
        gas: Nat,
        hops: Nat
```

El significado de los atributos es el siguiente:

- **mod**: reglas de la reescritura del objeto móvil
- **s:** estado actual
- **gas:** límite de recursos
- **hops**: número de saltos

Tanto el Cliente como el Servidor son de la clase RootObject y siempre tiene que haber uno de los dos por instancia del programa. La clase RootObject tiene la siguiente estructura

```
Class RootObject |
        cnt: Nat,
        guest: Set{Oid}
        forward : Map {Nat, Tuple{Loc, Nat}
        state : RootObjectState
        neighbors : Map  { Loc, Oid}
        defNeighbor : Default { Oid}
```

El significado de los atributos es el siguiente:

- **cnt:** contador para generar el nombre de objetos móviles
- **guest:** nombres de los objetos móviles que residen en el proceso
- **forward:** información donde se encuentran los objetos móviles
- **state:** estado (esperando conexión, activo)
- **neighbors:** asocia un socket a cada localización
- **defNeigbor:** socket por defecto

6. DESCRIPCIÓN DEL PROYECTO

Como hemos comentado anteriormente nuestro proyecto se basa en mostrar gráficamente la traza de un programa. Para ello hemos implementado una interfaz gráfica usando el lenguaje JAVA que interacciona con Maude y Mobile Maude.

Este proyecto servirá para tres tipos de uso:

- Una versión distribuida que consiste en ejecutar varias veces la aplicación y en cada una de ellas se cargará el programa a ejecutar y una instancia de éste. Estas instancias interaccionarán entre sí provocando que los objetos móviles viajen y, creando los estados y mensajes que mostraremos en la aplicación correspondiente junto con los objetos.

- Otra versión centralizada que permitirá realizar exactamente lo mismo que en la distribuida pero sólo ejecutando una vez la aplicación y cargando el programa a ejecutar. En la misma interfaz se mostraran las distintas instancias y los distintos estados, objetos y mensajes que se crean en cada instancia al viajar los objetos móviles

- Y una última versión que chequea modelos e indica si es cierto o falso. En el caso de que sea falso mostrará un contraejemplo

En los tres casos lo que se hace es analizar la traza que se va generando según interaccionan y viajan los objetos. Este análisis es dinámico ya que nunca sabemos cuando se acaba de recibir información.

Por eso la aplicación se queda escuchando y esperando siempre a recibir más información para ir analizando la traza que le llega y generar los estados correspondientes. La aplicación permite desplazarse desde el primer estado creado al último mediante la interfaz mostrando en rojo los cambios con respecto al anterior estado. También se puede ver la traza correspondiente al estado creado.

6.1. Descripción 1ª parte: Versión Distribuida

Inicialmente analizamos la traza que sale de ejecutar una instancia en un terminal. Como se muestra demasiada información, investigamos de qué forma podemos limitar esa traza para ir reduciendo su tamaño y así la dificultad al analizarla, quitando todo aquello que no nos es necesario mostrar.

Esto lo hacemos modificando el archivo mm-trace-options.maude. Tras esto, y ya disponiendo de una traza bastante más comprensible, comenzamos a diseñar la forma de analizarla para poder guardar de forma estructurada la información. Conseguimos también que la traza, en vez de salir toda de una vez, salga poco a poco, y que tengamos que ir ejecutando la orden "cont 1." para ir mostrando los estados. Esto nos facilita bastante su análisis ya que nos permite analizar paso a paso la ejecución. Vemos que los estados comienzan tras un "result Configuration", por lo que para ir capturando estados tendremos que ir cogiendo lo que haya entre una aparición de "result Configuration". En cambio, hay estados que no tenemos que tener en cuenta, puesto que corresponden a reglas que no nos interesan, por lo que los desecharemos y no lo guardaremos en el vector de estados.

Para saber qué estados evitaremos capturar, en las opciones de traza especificamos las reglas que queremos. Esto provoca que en los estados generados por las reglas que hayamos indicado, Maude escribe antes del "result" la regla correspondiente, y en los que no, no escribe nada. Así conseguimos saber qué estados hay que tener en cuenta y cuales no.

Una vez que tenemos el fragmento de traza que corresponde a un estado, tenemos que analizarlo para poder interpretar los objetos y los mensajes. Los objetos vienen encerrados entre "<" y ">". Dentro de un objeto, los atributos van en líneas distintas, por lo que se pueden interpretar fácilmente. Para saber el tipo de objeto, tenemos que comprobar lo que pone en la primera línea. Sobre los mensajes, es bastante más complicado, ya que aparecen en cualquier momento, y pueden aparecer varios seguidos. Además, hay de varios tipos.

Por tanto, tras tener ya una traza sobre la que trabajar, comenzamos el diseño de la aplicación. Una parte importante de la aplicación es la gestión de la unión de nuestra aplicación con Maude. Java permite ejecutar órdenes externas, por lo que no causa demasiados problemas. El problema es capturar la traza de Maude, ya que no se puede dejar a nuestra aplicación esperando a coger traza de Maude, porque no siempre se sabe si va a salir algo o se va a quedar esperando. Por tanto lo que hacemos es crear un hilo que va leyendo la traza, independiente del flujo general de la aplicación, y otro hilo que vaya interpretando esa traza. Así conseguimos que se pueda estar usando el programa (navegando por los estados, etc...) mientras estos 2 hilos se encargan de capturar la traza y de interpretarla. Cuando no hay traza disponible, los 2 hilos se quedan esperando a

que haya más traza. Si el motivo de que no haya más traza es que ya se ha mostrado un estado, se ejecuta "cont 1." para que Maude siga mostrando información.

Hay casos en los que la ejecución debe parar. Esto ocurre por ejemplo cuando intento ejecutar 2 veces la misma instancia, sin cambiar el puerto. La dirección por tanto será la misma, y si en Maude siguiéramos ejecutando "cont 1." Maude seguiría mostrando la misma información infinitas veces. Esto provocaría que el programa estuviera ejecutándose sin ningún sentido, ya que no podrá hacer nada al estar la dirección usada, y estará consumiendo recursos sin ninguna necesidad. Este caso no es único, y también ocurre cuando se cierra el servidor (los clientes reciben un ClosedSocket infinitamente), etc... En estos casos, habrá que evitar que el sistema siga intentando ejecutar la orden "cont 1." puesto que no serviría de nada.

El hilo lector es simplemente un hilo que va cogiendo la traza y guardándola en un vector, cada línea en una posición del vector. Cuando no hay más traza, se queda esperando, y comprobando cada cierto tiempo si hay nueva traza para coger.

El hilo intérprete es como una máquina de estados:

- En el estado 0, va pasando línea a línea, buscando un "result" que indique que comienza un nuevo estado. Cuando lo encuentra, busca la línea anterior, y si es una regla de las importantes, pasa al estado 1. En caso contrario, sigue en el estado 0. Si lo que encuentra no es una línea, sino un error de los anteriormente nombrados (dirección en uso, etc...) el sistema se para, y actualiza el interfaz notificando el problema.

- En el estado 1, va añadiendo las líneas a un vector temporal. Sigue en el estado 1 hasta que la información sobre un estado acaba. Una vez que llega a ese punto, y mediante métodos de la clase Utilidades, genera la información del estado usando el vector temporal en el que hemos ido añadiendo las líneas.

6.2. Descripción 2ª parte: Versión Centralizada

En la centralizada, se cargarán dentro de la interfaz tantas ventanas como instancias tenga el programa que ejecutamos. El mecanismo es el mismo que en la versión distribuida con la diferencia que tenemos todo en una misma interfaz (solo se ejecuta una vez la aplicación).

La versión centralizada tiene como estructura la versión distribuida, puesto que nos hemos basado en esta para diseñar esta parte, pero no es idéntica a la anterior; puesto que en la versión centralizada, al contrario que en la versión distribuida, solo se carga una vez el programa produciéndose las distintas ejecuciones de cada uno de los procesos que se producen en paralelo en una misma instancia.

En esta versión hemos reutilizado todo lo referente al análisis de mensajes y objetos, ya que independientemente de la versión, los objetos de Maude siguen teniendo los mismos atributos y la misma manera de representación. En cambio, hemos tenido que modificar el análisis de la traza en varios campos:

- A nivel más interno, es decir, referente a objetos y mensajes no se ha modificado debido a que como ya hemos comentado es independiente de la versión la representación de los objetos y mensajes.

- Hemos creado un nuevo análisis a nivel intermedio, es decir, la parte que analiza la traza perteneciente a cada proceso y que encapsula los distintos objetos y mensajes de ese proceso; ya que esta traza es totalmente nueva si la comparamos con la versión distribuida por lo que ha requerido de este nuevo análisis para poder mostrar gráficamente cada uno de los procesos.

- Hemos creado un nuevo análisis a nivel más externo, es decir, la parte que analiza la traza que encapsula las descripciones de cada uno de los procesos que se ejecutan simultáneamente; debido a que en la versión centralizada Maude solo genera una traza por programa dentro de la que se incluyen, por así decirlo, distintas subtrazas correspondientes a estos distintos procesos.

También hemos tenido que realizar una parte de aprendizaje de nuevas partes del lenguaje Java, ya que ninguno de los componentes del grupo teníamos idea de cómo programar distintos Frameworks simultáneos para mostrar en cada uno de forma independiente cada uno de los procesos que se ejecutan en un programa de Maude en la versión centralizada; por lo que podemos indicar que tras este estudio hemos conseguido que cada proceso de los que se ejecutan en una misma instancia de la versión centralizada se muestre en una ventana independiente. Cada ventana muestra los objetos y mensajes propios de cada proceso, pudiendo redimensionar dicha ventana a gusto del usuario.

Se pueden ver una serie de similitudes con la versión distribuida como son:

- Los distintos pasos a seguir para ejecutar un programa como son cargar Maude, cargar el programa o cargar la instancia.

- El manejo de la aplicación prácticamente es el mismo salvo que en esta versión se puede disfrutar de todos los procesos de un programa en una misa interfaz gráfica, pudiendo sacar la traza de cada proceso en cada estado (como en la versión distribuida).

- En cada estado se puede observar resaltados en color rojo los distintos atributos que hayan cambiado de valor respecto al estado anterior.

- Se pueden modificar las reglas que se consideran oportunas a la hora de crear un nuevo estado; sin embargo, el cambio en la elección de las reglas ahora se aplicaría totalmente a todos los procesos de la instancia, aunque cada proceso disponga de una ventana independiente, al contrario que en la versión distribuida que al tener cada instancia una interfaz gráfica independiente, se podía elegir en cada una las reglas creadoras de estados.

6.3. Descripción 3ª parte: Model Checker

Model checker es el nombre de la tercera versión de nuestra aplicación. Esta versión consiste en la verificación de ejemplos de instancias de programas Maude. Dichos ejemplos pueden ser ciertos en cuyo caso nuestra aplicación solo mostraría el resultado de forma gráfica dejando de forma transparente para el usuario el análisis de la traza; o bien puede ser falso en cuyo caso no solo mostraríamos también de forma gráfica el resultado del ejemplo sino que además mostraríamos gráficamente los distintos estados de la misma forma que en la versión centralizada del contraejemplo que demuestra la falsedad del ejemplo que se quiere probar.

7. SEGUIMIENTO DEL PROYECTO

Para facilitar más el entender el desarrollo del programa hemos decidido explicar paso a paso lo que hemos ido haciendo para el proyecto, reuniones, documentación, desarrollos, etc.…

A la hora de implementar el proyecto primero desarrollamos la versión Distribuida, y luego adaptamos lo que teníamos a la versión Centralizada. Una vez que más o menos estaban las dos de forma estable fuimos corrigiendo errores de forma paralela en ambas versiones.

Estos son los pasos que hemos dado para implementar las distintas partes. En los siguientes puntos los explicaremos con más detalle.

7.1. Seguimiento 1ª parte: Versión Distribuida

- Reunión toma de contacto
- Leer documentación sobre Maude y Mobile Maude
- Reunión de especificación de requisitos
- Búsqueda de información sobre Maude y Mobile Maude
- Lectura de documentación sobre comandos de Maude
- Instalación de Maude
- Prueba ejemplos Maude
- Lectura de documentación sobre trazas
- Familiarización con las trazas de Maude
- Reunión para establecer las características de la interfaz gráfica
- Análisis detallado del ejemplo printers
- Desarrollo de un primer programa para ver cómo conectar Java con Maude
- Especificación de cómo analizar la traza
- Realización de la gramática que define los distintos estados
- Comunicación de la aplicación con Maude
- Creación del hilo lector
- Definición del vector de traza

- Creación del hilo intérprete
- Obtención de la información de un estado
- Definición del vector de estados
- Desarrollo de una 1ª versión de la interfaz gráfica
- Interpretación de los mensajes
- Creación del hilo que ejecuta el DownTerm
- Destacar los cambios en atributos con respecto al anterior estado
- Eliminación del estado inicial
- Mejoras en la interfaz
- Introducir botón de ver traza
- Reunión de seguimiento
- Archivo de propiedades. Configuración de la ruta de Maude
- Posibilidad de elección de las reglas a mostrar
- Muestra del estado 1 del vector automáticamente
- Reunión de seguimiento
- Control de errores
- Pruebas y arreglo de errores

7.2. Seguimiento 2ª parte: Versión Centralizada

- Reunión de requisitos
- Generación de otro proyecto rehusando el código
- Cambio en el interprete y en la representación de estados
- Vector de Terminales
- Cambio en la interfaz gráfica
- Mejoras en la interfaz
- Destacar los cambios en atributos con respecto al anterior estado
- Introducir botón de ver traza
- Reunión de seguimiento
- Archivo de propiedades. Configuración de la ruta de Maude
- Elección de las reglas que a mostrar

- Muestra del estado 1 del vector automáticamente
- Reunión de seguimiento
- Control de errores
- Pruebas y arreglo de errores

7.3. Seguimiento 3ª parte: Model Checker

- Reunión de seguimiento
- Generación de otro proyecto reutilizando código
- Cambios en el intérprete
- Cambios en la interfaz gráfica.
- Modificación en la forma de capturar reglas
- Control de errores
- Prueba y error

7.4. Unión

- Reunión de seguimiento
- Reestructuración del código
- Creación de una nueva ventana inicial del proyecto.
- Optimización del uso del vector de traza.
- Implementación de la opción de resetear sistema.
- Prueba y error
- Comentarios el código y JavaDoc

8. VERSIÓN DISTRIBUIDA

Como hemos descrito en el punto anterior los pasos que hemos seguido son los siguientes

8.1. Reunión toma de contacto

A principios de Octubre nos reunimos con nuestro Tutor del Proyecto para que nos explicara en que consistía el proyecto que íbamos a desarrollar. En esta reunión nos explicó un poco la idea del proyecto, en qué consistía, qué programa se iban a usar, cuál era la finalidad del proyecto, etc… En esta reunión nos comentó que iba a ser una interfaz gráfica desarrollada en JAVA que mostrara la traza de la ejecución de un programa para Mobile Maude de manera más visual y legible que la traza sacada mediante la ejecución en un terminal.

Como no conocíamos que era Mobile Maude ni Maude se nos facilitó dos documentos a modo de información y ejemplo que son "*Playing with Maude"* y "*A distributed implementation of Mobile Maude"*.

8.2. Leer documentación sobre Maude y Mobile Maude

Antes de especificar exactamente los requisitos del proyecto, necesitábamos información sobre Maude y Mobile Maude y en que consistía este lenguaje. Para ello nos leímos la información facilitada por el Tutor (documentos *Playing with Maude"* y "*A distributed implementation of Mobile Maude"*) en la que nos explicaba en que consistían y las principales características que nos serviría como nociones básicas para plantear el desarrollo del proyecto.

8.3. Reunión de especificación de requisitos

Volvimos a tener otra reunión con nuestro Tutor en el que nos explicó con más detalle los requisitos del sistema. Nos comentó que el proyecto iba a constar de varias partes, en las cuales habría que analizar la traza de un ejemplo de Mobile Maude y mostrar gráficamente los atributos de los objetos y los mensajes que se envían. Se quedó en otra próxima reunión en la que se decidiría con exactitud cómo sería la distribución de esos atributos y qué reglas se iban a usar.

8.4. Búsqueda de información sobre Maude y Mobile Maude

Antes de la siguiente reunión nos pusimos a buscar información en Internet sobre Mobile Maude y Maude y consultar la página oficial de Maude que es:

http://maude.cs.uiuc.edu

Esta página contiene un manual bastante amplio sobre Maude que puede ser de gran utilidad.

8.5. Lectura de documentación sobre comandos de Maude

Una vez que ya tuvimos una noción más clara de Maude nuestro Tutor nos proporcionó un documento con los comandos básicos de Maude llamado "*Complete List of Maude Commands"*.

8.6. Instalación de Maude

El siguiente paso fue descargarse de la página de Maude (http://maude.cs.uiuc.edu) la última versión de Maude e ir familiarizándonos con los comandos que antes habíamos leído en la documentación.

8.7. Prueba de ejemplos Maude

Tras probar un poco los comandos probamos alguno de los ejemplos que pueden descargarse de la página de Maude. También probamos el ejemplo de las impresoras, el cual se tiene que ejecutar en 3 terminales distintos y vimos cómo la traza varía según se ejecuta otras instancias y según van viajando los objetos.

8.8. Lectura de documentación sobre trazas

Como la traza de la ejecución es bastante extensa y muchas cosas iban a ser innecesarias para sacar la información que queríamos gráficamente, nuestro Tutor nos facilitó otro documento (*"Debugging and Troubleshooting")* en que se explicaba la traza de Maude y los comandos que podían facilitar nuestra tarea, como por ejemplo, quitar cosas de la traza que no nos fuesen útiles, mostrar la ejecución paso a paso, seleccionar las reglas que queremos mostrar….

8.9. Familiarización con las trazas de Maude

Tal y como hemos hecho en los otros casos que se nos ha facilitado información nueva que nos pueda servir para desarrollar nuestro proyecto, estuvimos probando las distintas opciones que nos ofrecen los comandos y ver qué combinación es la que más se adecúa a nuestras necesidades.

8.10. Reunión para establecer las características de la interfaz gráfica

Un mes después de la primera reunión aproximadamente, y en el cual estuvimos familiarizándonos con el entorno Maude, nos volvimos a reunir con nuestro Tutor para ver qué características tenía que tener la interfaz, qué objetos se querían pintar, qué atributos de los objetos.

En esta reunión se acordó que se mostraran los objetos de la clase MobileObject y el objeto de la clase RootObject que se haya creado (ya sea ClientRootObject o ServerRootObject).

Los objetos de la clase RootObject los representaremos de la siguiente manera:

Nombre clase	
mod	guests
forward	state
neighbors	defNeighbor

Los objetos de la clase MobileObject los representaremos de la siguiente manera:

Nombre clase	
mod	s
gas	hops

Para sacar el valor correspondiente a cada atributo de las dos clase de objetos se sitúa el cursor sobre el atributo y saldrá un mensaje con el valor de ese atributo.

Las reglas relevantes que tendremos en cuenta a la hora de la traza serán las siguientes:

- go-loc
- go-find-loc
- arrive-loc
- arrive-find-loc
- arrive-find-proc
- msg-send
- msg-arrive-to-loc
- msg-in
- msg-out-to
- msg-out-go
- msg-out-go-find
- do-something
- create-object
- message-out-kill
- changeSend

Los mensajes que mostraremos serán los siguientes:

- go-find
- go
- to_hops_in
- to
- Send

8.11. Análisis detallado del ejemplo printers

Una vez que tenemos los atributos de los objetos y las características que tenemos que mostrar nos pusimos a analizar el ejemplo printers para ver qué información nos daba Maude y cómo podíamos analizar esa información, es decir, ver cómo es la traza, ver qué características y situaciones se cumplan siempre y así nos sirviera para poder saber qué es lo que vamos a analizar, si llega un nuevo objeto, si es un mensaje, si es una regla que nos interesa o no.

De esta manera viendo las cosas comunes, nos ha servido para poder pensar la forma en que íbamos a sacar la información de la traza y ubicarla en su sitio correcto. El problema es que mucha de las instrucciones que contienen la traza no es relevante y eso hace que dificulte la legibilidad a la hora de analizarlo.

8.12. Desarrollo de un primer programa para conectar Java con Maude

Antes de empezar a desarrollar el código decidimos que mejor era empezar etapa a etapa por lo que decidimos probar con un programa sencillo la forma de conectar Java con Maude.

La forma que hemos encontrado ha sido usar una instrucción de Java que es Runtime.getRuntime().exec(). Con esta instrucción conseguimos ejecutar Maude desde Java, pero simplemente crear el proceso.

Para nuestra aplicación, no es suficiente con crear el proceso y ejecutarlo, sino que hay que comunicarse con él, ya que tendremos que ejecutar distintas órdenes para cargar los archivos, hacer que la ejecución continue, etc...Para ello usamos las

posibilidades que da la clase Process, definida en el API de Java 1.5. En esta clase disponemos de los métodos getInputStream() y getOutputStream(). Mediante estos métodos podemos obtener dos streams que nos permiten obtener la salida del proceso (lo que mostraría en la consola) y mandar instrucciones al proceso (lo que se introduciría en la consola). De momento, en este primer programa, solo creamos el proceso y vemos mediante las herramientas de gestión de Linux que el proceso se crea correctamente.

8.13. Especificación de cómo analizar la traza

Para sacar la traza de la manera apropiada y eliminando la información que no nos sirve usamos

```
set trace on .
set print conceal on .
print conceal mod_is_sorts_._____endm .
print conceal Send send .
set trace condition off .
set trace substitution off .
set trace body off .
set show advisories off .
set trace builtin off .
set trace whole off .
set trace mb off .
set trace eq off .
set trace rewrite off .
set trace select on .
trace select arrive-loc arrive-find-loc msg-send msg-
arrive-to-loc  msg-in message-out-to message-out-go
message-out-go-find do-something create-object  message-
out-kill redirect redirectDef Received .
```

Con estas opciones conseguimos limpiar la traza lo suficiente como para dejar la información que necesitamos para así facilitar la forma de construir la gramática, la máquina de estados y el intérprete.

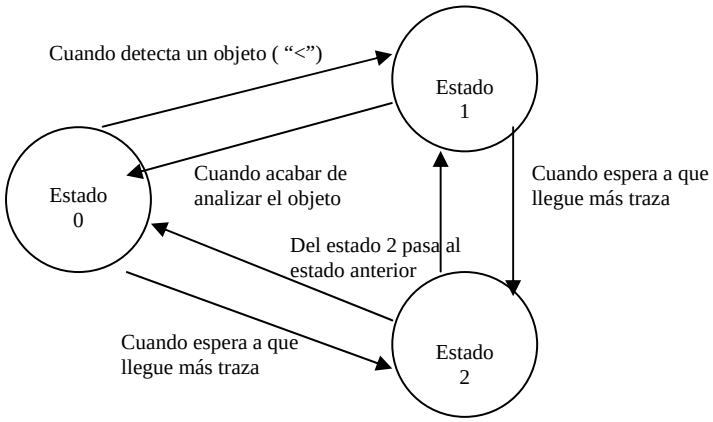
8.14. Realización de la Gramática que define los distintos estados

Al analizar la traza nos dimos cuenta que hay ciertas combinaciones que siempre se repiten, en cuanto a la forma de empezar o acabar un objeto, la forma de interpretar las reglas, por lo que decidimos crear una gramática para diferenciar entre los distintos estados que se crean.

Un objeto viene siempre encerrado entre "<" y ">". Dentro de este objeto los atributos van en líneas distintas (aunque un atributo puede ocupar más de una línea). En la 1ª línea del objeto sacamos qué tipo de objeto es.

La máquina de estados está compuesta por tres estados:

- **estado 0** -> sigue leyendo sin hacer nada
- **estado 1** -> llega a este estado cuando detecta un objeto
- **estado 2** -> cuando no llega mas traza



8.15. Comunicación de la aplicación con Maude

Como hemos dicho anteriormente, para el objetivo de la aplicación no es suficiente con ejecutar el proceso de Maude, sino que hay que comunicarse con él, ya que necesitamos tanto enviarle información (indicarle los ficheros que tiene que cargar para cargar Mobile-Maude y el programa o modelo que queramos cargar) como recibirla (recibir la traza que Maude muestra). Para ello, la clase Process nos permite obtener un OutputStream y un InputStream del proceso. El OutputStream nos servirá para enviarle al proceso las instrucciones, y el InputStream para recibir la traza. Creamos por tanto un BufferedWriter usando el OutputStream del proceso, y así

conseguimos que haciendo un write sobre el BufferedWriter con la orden que deseemos ejecutar, el proceso reciba la instrucción y la ejecute.

En cuanto a sacar la traza con el InputStream, el proceso no es tan sencillo. Creamos un BufferedReader con el InputStream, y de ahí ya podemos leer la traza obtenida. El problema es que nosotros no sabemos la longitud de la traza, ni cuando va a estar disponible. Si en el BufferedReader intentamos leer algo y no hay nada, el BufferedReader bloquea la ejecución del programa hasta que le llega algo para leer. Este comportamiento no es admisible en nuestra aplicación por lo que tenemos que estudiar diversas alternativas.

8.16. Creación del hilo lector

Para evitar que nuestra aplicación se quede bloqueada esperando traza, creamos otro hilo a parte del flujo principal de la ejecución. Así el análisis de la traza, el uso del programa, y la captura de la traza podrá ser simultáneo. En este hilo, simplemente estamos en un bucle infinito, que va comprobando si el BufferedReader esta ready (hay algo para leer), y en ese caso lo lee. En caso contrario, se hace un sleep durante un determinado tiempo, hasta que se vuelve a probar.

8.17. Definición del vector de traza

Una vez que vamos capturando la traza, tenemos que analizarla. Para ello, necesitaremos otro hilo ya que tampoco podríamos dejar bloqueado el programa mientras la analizamos. Para que la traza que capturamos en un hilo la podamos usar en otro para analizarla, usamos un vector común a los 2 hilos. El vector es una estructura de Java que está a salvo de problemas de concurrencia, por lo que no tenemos que gestionarla nosotros. Definimos el vector como vector de String. El hilo lector va guardando una línea de traza en cada posición del vector.

8.18. Creación del hilo intérprete

Este hilo se encargará del análisis de la traza, por lo que podemos decir que es la parte fundamental de la aplicación. La traza la tiene guardada en el vector de String que hemos comentado anteriormente. Para analizarla, usamos la máquina de estados comentada anteriormente, pero desarrollándola más en profundidad.

Al comenzar a analizar la traza, el estado en que se encuentra el intérprete es el 0. Se va leyendo línea a línea sin cambiar de estado al intérprete, hasta que encontramos una línea que comienza por la palabra “result”. Esto nos indica que las siguientes líneas corresponden a la información de un nuevo estado. Pero no hay que tener en cuenta todos los estados, solo aquellos a los que se ha llegado tras aplicar una regla determinada. La regla aplicada va siempre justo 2 líneas antes del result, por lo que observamos lo que hay en esa posición y vemos si corresponde a una de las reglas que hay que analizar. En ese caso, habrá que pasar al estado 1 del intérprete.

Una vez en el estado 1, vamos leyendo líneas y guardándolas en un vector temporal, hasta que la información sobre un estado acaba. Saber cuándo ocurre esto no es tarea fácil ya que la traza de Maude no es completamente regular, y hay bastantes excepciones. Pero fundamentalmente ocurre cuando una línea comienza por “Maude>”, por “rewrites” o por asteriscos (salen cuando se aplican reglas condicionales). Una vez que se llega a este punto, tenemos en un vector temporal la información del estado correspondiente. Este vector es el que analizamos para crear las estructuras que nos representan la información del estado. Pasamos al estado 0, para que el análisis de la traza siga su curso.

Existe la posibilidad de que en mitad del análisis, deje de haber traza, porque Maude este aún calculando algo o porque el hilo lector aún no la haya capturado. Para eso usamos el estado 2. Dirigimos el autómata a ese estado cuando por algún motivo no hay más traza que analizar. Este estado se queda esperando, y cuando vuelve a haber información nueva en el vector vuelve al estado en el que se encontraba el autómata antes de pasar a él.

También existe la posibilidad de que deje de haber traza para analizar porque el hilo lector ya ha reconocido toda la traza disponible para los pasos ejecutados sobre el programa de Maude; es entonces cuando ejecutamos “cont 1” sobre Maude para pasar al siguiente paso en la ejecución.

Tenemos por tanto ya un proceso para conectarse con Maude, un hilo para escribir las instrucciones y poder ejecutarlas, un hilo que captura la traza y otro hilo que va analizando esa traza y que va metiendo en vectores temporales la información de los estados. Ahora debemos por tanto coger esos vectores temporales y obtener la información que necesitamos del estado.

8.19. Obtención de la información de un estado

Una vez que tenemos la información sobre un estado en un vector, debemos analizarlo para obtener la información que nos interesa. Este es uno de los pasos más

complicados ya que dentro de un estado puede haber mensajes y objetos, y los mensajes pueden aparecer en cualquier lugar, a veces varios seguidos, etc... Todo esto hace que el análisis sea más complejo.

En la traza, la información de un estado es de la siguiente forma:

<
mensajes
<objeto>
mensajes
<objeto>
...
<objeto>
mensajes
>

Por tanto al analizarlo tenemos que tener en cuenta esta estructura. Vamos interpretando mensajes (puede haber uno o varios) hasta que una línea comienza por "<". En ese caso, estaremos ante un objeto. El objeto puede ser de varios tipos. El tipo lo sabemos analizando la primera línea. Vamos guardando la información sobre ese objeto en un vector temporal, y cuando llegamos a una línea que empiece por ">" será porque estamos al final del objeto. Ese vector temporal lo analizamos de distinta forma dependiendo del tipo de objeto que sea.

Los mensajes van entre paréntesis. Por ello, para analizarlo lo que hacemos es capturar desde un paréntesis de apertura hasta el paréntesis de cierre, y todo lo que haya entre ellos dos forma parte de un mensaje. Hay mensajes de varios tipos, y cada uno se interpreta de distinta forma, pero de momento simplemente capturamos la parte de la traza correspondiente al mensaje, sin tratarla de ninguna forma.

Para los objetos, la estructura es distinta dependiendo del tipo de objeto, pero se ajusta a un modelo general, como este:

<nombre: tipo objeto
atrib1:valor,
atrib2:valor,
.
.
.
>

Creamos dos métodos, uno para cada tipo de objeto (home o mobile-object), y dependiendo del valor de la primera línea usamos un método u otro. La estructura de los métodos es bastante sencilla, simplemente van línea a línea capturando y guardando tanto el nombre del atributo como el valor.

8.20. Definición del vector de estados

Para guardar los distintos estados hemos creado un vector en el que en cada posición guardamos un estado. Este vector de estados es el que nos deja ir moviéndonos por la traza, para mostrar los distintos estados mediante la interfaz gráfica, así de esta forma podemos movernos en ambas direcciones. Cada vez que nos encontremos en una posición del vector analizamos el contenido y mostramos los valores de los atributos correspondientes en la interfaz.

8.21. Desarrollo de una 1ª versión de la interfaz Gráfica

Para probar que el programa va funcionando correctamente, necesitamos avanzar en el desarrollo de la interfaz gráfica, para poder movernos por el vector de estados y observar que interpretamos y guardamos correctamente la información.

Esta primera versión es muy sencilla, y simplemente sale toda la información en labels, en modo texto, y ponemos un botón avanzar y otro retroceder, para poder movernos por el vector de estados. Llevamos en una variable la posición mostrada en cada momento, y mediante los botones de avanzar y de retroceder se aumenta o disminuye cuando sea posible esa variable, y se actualiza la vista.

Para poder cargar las aplicaciones y las instancias, se crea un menú a través del cual se puede cargar Maude y Mobile Maude, la aplicación que deseemos y la instancia que queramos.

8.22. Interpretación de los mensajes

Actualmente nuestra aplicación no trata los mensajes, simplemente los capta y los muestra, lo que produce que la información mostrada sobre los mensajes no sea del todo clara. Para ello, se incluye una gestión de los mensajes.

Nuestra aplicación debe tener en cuenta cuatro de los distintos tipos de mensajes que pueden aparecer en la traza: go-find, go, to-hops-in y to. Cada mensaje tiene un formato distinto, por lo que para analizarlo habrá que hacerlo de distinta forma. En general, usamos un método que nos permite obtener el texto que hay entre dos caracteres que nosotros le indiquemos. Eso nos permite ir obteniendo los distintos campos importantes de los mensajes, que varían en función del mensaje, variando los caracteres de inicio y fin.

8.23. Creación del hilo que ejecuta el DownTerm

En la traza, el valor del atributo S tiene un formato “incomprensible”:

```
('_&_['<_:_|_>['o['l['"127.0.0.1".String,'0.Ze
ro],'s_['0.Zero]],'Invitation.Invitation,'_`,_
['answer`:_['maybe.Maybe`{Answer`}],'center`:_
['o['l['"127.0.0.1".String,'0.Zero],'0.Zero]],
'guest`:_['o['l['"127.0.0.1".String,'s_['0.Zer
o]],'0.Zero]],'invitationSt`:_['travelling.Inv
itationState],'items`:_['_`,_['item1.Item,'ite
m2.Item,'item3.Item]]]],'go-
find['o['l['"127.0.0.1".String,'s_['0.Zero]],'
0.Zero],'l['"127.0.0.1".String,'s_['0.Zero]]]]
)
```

Esta información es ilegible para el usuario, por lo que sería inútil que nuestra aplicación mostrase esto. Mobile Maude ofrece una instrucción que nos permite obtener esta información de forma más legible. Esta instrucción es DownTerm. Si en un terminal cargamos Maude, Mobile-Maude y la aplicación, y ejecutamos “red downTerm(parámetro, none & none).”, siendo parámetro el valor que la traza mostraba para S, en el terminal nos sale como resultado la información de forma mucho más legible. La ejecución del downTerm devolvería información con el formato siguiente:

< o(l("127.0.0.1", 0), 0) : Center |

items : (item1, item2, item3),

agents : empty,
accepted : empty,

buyers : (o(l("127.0.0.1", 1), 0), o(l("127.0.0.1", 2), 0)),

centerState : waitingResponse,
counter : 0,

bestOffer : maybe,

bestAgent : maybe,

current : maybe

> & newo(mod_is_sorts_._____endm(...),< tmp-id : Invitation |guest : o(l("127.0.0.1", 1), 0),

answer : maybe,

center : o(l("127.0.0.1", 0), 0),

invitationSt : travelling,

items : (item1, item2, item3)

>, tmp-id) newo(mod_is_sorts_._____endm(...),< tmp-id : Invitation |

guest : o(l("127.0.0.1", 2), 0),

answer : maybe,

center : o(l("127.0.0.1", 0), 0),

invitationSt : travelling,

items : (item1, item2, item3)

, tmp-id)

Esta es la información que guardamos y que posteriormente mostraremos. Para obtenerla, necesitamos una estructura similar a la que hemos diseñado para obtener la traza principal. Necesitamos un proceso nuevo, un hilo lector y un BufferedWriter para escribir las instrucciones. Toda esta estructura es transparente al usuario, que lo único que observa es que el valor del atributo S tiene el formato correcto.

8.24. Destacar los cambios en atributos con respecto al anterior estado

Los valores de los atributos de los objetos tienen a menudo un tamaño bastante grande. Esto, añadido al hecho de que para ver esos valores hay que colocar el ratón encima del atributo correspondiente, hace que no sea fácil observar los cambios en los valores al cambiar de un estado a otro. Una posible solución sería mostrar los valores directamente, pero algunos son excesivamente grandes como para mostrarlos siempre.

Por tanto decidimos que la mejor opción es indicar de alguna forma los atributos que han cambiado su valor respecto al que tenían en el estado anterior.

Para ello, al crear la representación gráfica de los objetos y sus atributos, vamos comparando los valores que tienen con los que tenían en el estado anterior, y resaltamos con un color llamativo aquellos que hayan cambiado.

Esto permite observar rápidamente los atributos que han cambiado, y así no tener que ir recorriendo con el ratón todos los atributos para darse cuenta de cuales cambian y cuales no.

8.25. Eliminación del estado inicial

Actualmente capturamos el estado inicial, que es el primer estado que aparece en la traza. El problema es que el valor de los atributos en este estado no tiene el mismo formato que en resto de estados, lo que puede provocar confusiones al analizar el flujo global. Por este motivo, y porque los valores que se capturan en este estado pueden observarse en el fichero de la instancia, decidimos no capturar ni mostrar este estado.

Para conseguirlo tenemos que cambiar el intérprete para que no pase al estado 1 la primera vez que cree que va a capturar un estado.

8.26. Mejoras en la interfaz

En este momento mostramos la información en modo texto. Para facilitar la comprensión al usuario, diseñamos una nueva forma de mostrar los estados, objetos y atributos. Lo hacemos con cajas, de forma que un objeto es una caja compuesta de un conjunto de cajas, una para cada atributo. Para ver el valor de cada atributo hay que colocar el ratón encima del atributo que deseemos.

Con este nuevo diseño gráfico se observa de forma rápida el proceso de creación/eliminación de objetos, ya que las cajas aparecen y desaparecen.

8.27. Introducir botón de ver traza

Nuestra aplicación permite ver los objetos que hay en cada momento y los valores que toman los atributos de esos objetos. Pero también queremos que el usuario pueda ver la traza exacta que ha sacado Maude, y así comprobar que el resultado es el correcto y poder ver algunos mensajes y algunos datos que no se tienen en cuenta para la aplicación por no tener la suficiente relevancia.

Para dar al usuario esta posibilidad, creamos un nuevo atributo en la clase Estado, en el cual guardamos la porción de traza correspondiente a la información de

ese estado. En la pantalla principal de la aplicación mostramos ahora un botón que al pulsarlo provoca que se abra una nueva ventana en la que observar esta traza.

8.28. Reunión de seguimiento

Programamos otra reunión de seguimiento, para ver si el desarrollo de la aplicación va por buen camino y las cosas que se podrían o deberían cambiar.

En esta reunión, sacamos algunas conclusiones:

- La forma de cargar Maude y Mobile Maude (cada vez que se abre la aplicación hay que especificar la ruta) se debe mejorar para hacer más cómodo y sencillo el uso de la aplicación el programa.
- Sería conveniente que la aplicación dispusiera de algún tipo de filtrado de reglas. Este filtrado provocaría que al avanzar o retroceder sólo se tuvieran en cuenta las reglas que el usuario hubiera marcado. Así, si el usuario sólo quiere ver los estados a los que se llega tras la aplicación de “do-something” debería marcar solo esa regla, y esto provocaría que al pulsar sobre avanzar/retroceder no se avanzaría/retrocedería solo un estado, sino los necesarios hasta llegar a un estado cuya regla aplicada sea “do-something”.
- El primer estado que se capture debería aparecer instantáneamente, y no esperar a que el usuario pulse el botón de avanzar.
- No se sabe cuando la aplicación esta parada o capturando traza. Se debería mostrar esta información de alguna forma.

8.29. Archivo de propiedades. Configuración de la ruta de Maude

La forma de cargar Maude y Mobile-Maude no es todo lo cómoda que cabría esperar. Hay que buscar el archivo ejecutable de Maude y el de Mobile-Maude cada vez que se abre la aplicación.

Para evitar esto, creamos un fichero de propiedades. Este fichero servirá para configurar el programa. En él, debemos especificar la ruta de Maude y la de Mobile-

Maude, y al ejecutar la aplicación simplemente tendremos que pasar como parámetro la ubicación de este fichero de propiedades.

Así conseguimos que el usuario no tenga que especificar en cada ejecución las rutas, sino que la aplicación las coge automáticamente del fichero de propiedades. El fichero se ajusta a las especificaciones de los ficheros de propiedades en modo texto:

rutaMaude = /home/maude-linux/maude.linux

rutaDistMobile = /home/mobile-maude/mm-code/mobile-maude.maude

Además, ahora se carga a la vez Maude y Mobile-Maude, puesto que no tiene sentido la carga de uno sin el otro. Se añaden shortcuts al menú que permiten cargar Mobile y Mobile-Maude pulsando simplemente un par de teclas.

8.30. Posibilidad de elección de las reglas a mostrar

Como acordamos en la reunión anterior de seguimiento se daría al usuario la posibilidad de elegir las reglas a partir de las cuales se llegaría a los estados que se van a mostrar tras esa elección.

Entre los miembros del grupo llegamos a la conclusión de poner una opción en el menú en la que dábamos la posibilidad al usuario de elegir mediante “Check boxes” las diferentes reglas disponibles.

8.31. Muestra del estado 1 del vector automáticamente

Como una de las conclusiones de la última reunión de seguimiento hecha hasta el momento se decidió que se mostrara automáticamente el primer estado capturado. Para ello, primeramente, en la clase que implementa la pantalla principal se crea e instancia un objeto de la clase que implementa el interfaz de tipo Runnable. Una vez instanciado este objeto se implementa el método run para que mediante este método se ejecute la misma acción que si el usuario pulsa el botón siguiente, es decir, llamamos al método doClick() de nuestro botón siguiente.

Finalmente en el Intérprete usamos la instrucción SwingUtilities.invokeLater para llamar al objeto que creamos anteriormente en la clase que implementa la pantalla

principal y ejecutar su método run en un hilo independiente para que esto no interfiera en la ejecución de los demás hilos de la aplicación.

8.32. Reunión de seguimiento

El la última reunión de seguimiento de la versión distribuida el tutor nos dio consejos para dejar cerrada esta versión. Estos consejos fueron el control de errores de la versión distribuida mediante estructuras “Try and Catch” y realizar una serie de pruebas para observar si la aplicación tenía algún fallo en el análisis de la traza y poder solucionarlos antes de cerrar la aplicación.

8.33. Control de errores

Realizamos sobre la aplicación una estructura “Try and Catch” para la captura de excepciones. Esta estructura captura errores inusuales que puedan darse en la aplicación a modo de excepciones. Estas excepciones se tratan de forma que se escriben en el log para su posterior observación y estudio por parte de los desarrolladores de la aplicación.

8.34. Pruebas y arreglo de errores

Esta parte del desarrollo de la aplicación es la que se conoce como fase de “Prueba y error” y consiste en que gracias a los diversos ejemplos de ejecución distribuida facilitados por el tutor pudimos probar y comprender las diversas irregularidades en la traza de Maude para ver y corregir los fallos que inicialmente habíamos cometido en este análisis de la traza.

9. VERSIÓN CENTRALIZADA

9.1. Reunión de requisitos

Poco después de finalizar el primer cuatrimestre tuvimos una nueva reunión con el tutor del proyecto en la cuál se nos especificó los requisitos principales de esta nueva versión. Se nos procuraron diversos ejemplos de esta nueva versión para que pudiéramos analizar la traza y concluir los nuevos estados, y se nos explicó el funcionamiento básico para facilitarnos el análisis.

9.2. Generación de otro proyecto rehusando el código

Ya que en la versión distribuida habíamos tratado directamente con Maude y que ya sabíamos la estructura y comportamiento de los objetos que componen las aplicaciones; partimos de esta base para poder reutilizar la mayor parte del código encargado de analizar la traza de la versión distribuida en esta nueva versión, y del código de la interfaz gráfica para así crear un conjunto de mini aplicaciones similares. Las partes que conseguimos reutilizar fueron:

- El código encargado de reconocer los objetos que componen las aplicaciones de Maude desde los MobileObject hasta los RootObject con sus diferentes variedades.

- El código de la interfaz gráfica, ya que el funcionamiento prácticamente es el mismo, salvo que ahora mostrábamos los diversos procesos que componen una instancia en la misma interfaz gráfica por lo que necesitamos la ayuda de contenedores que desconocíamos en la versión distribuida.

- El código encargado de la creación y el tratamiento de los objetos así como el código encargado de colorear los atributos que se modifican en cada objeto de un estado a otro.

- El código que genera la visualización de la traza de cada estado con la diferencia de que en esta nueva versión se dispone en cada estado de “x” visualizaciones, una por cada uno de los procesos que componen la instancia del programa.

- El código encargado de la creación y actualización del “logger”, con la diferencia de que ahora se “loggean” todos los procesos que componen la instancia en el programa en el mismo archivo al estar todos comprendidos dentro de la misma interfaz gráfica.

9.3. Cambio en el intérprete y en la representación de estados

Cuando empezamos a implementar la versión centralizada nos encontramos que el formato de la traza de Maude en la versión centralizada es distinto al que teníamos para la versión distribuida.

Además de esto, tendríamos que hacer otros cambios en el intérprete y la representación de estados ya que necesitamos nuevas estructuras para guardar la información Esto se debe a que antes con un vector de estados era suficiente, pero no en la versión centralizada, ya que en cada ejecución no tenemos sólo la información de un Terminal, sino la de varios. Con Terminal nos referimos a lo que Maude nos indica como pid(x), siendo x el identificador numérico.

La traza que muestra Maude para la versión centralizada es de la siguiente forma:

regla

información no necesaria

result...

información no necesaria

< pid(0) : . . .

Información sobre el estado, sigue el mismo formato que la información sobre un estado de la versión distribuida

mensajes

<objeto>

mensajes

<objeto>

...

<objeto>

mensajes

...

>

< pid (1) : . . .

Así se procedería con todos los “pid” hasta que finalice el último entonces se seguiría pasando información (que sería no necesaria) hasta el siguiente result y

entonces se retrasaría el interprete hacia la regla de este result para empezar el análisis desde hay otra vez.

Por lo tanto, aunque hay partes que son parecidas a la traza de la versión distribuida, el autómata tiene que ser rediseñado.

En el nuevo autómata, el estado 0 va pasando línea a línea hasta que se encuentra una línea con la palabra result, lo que indica que comienza la información sobre un nuevo estado para todos los terminales. En ese momento, se pasa al estado 3, en el cuál se consumen líneas hasta que se encuentra una línea que empieza por "pid(x)", esto indica que comienza la información sobre el estado del Terminal x; siendo x, el número que va entre paréntesis, el identificador del proceso. Tras finalizar esto el autómata pasa al estado 1 que simplemente va guardando las líneas en un vector temporal para su posterior análisis. Son guardadas las líneas siguientes hasta que el intérprete encuentra una línea que comience por "pid(x+1)" (en ese caso habrá acabado la información sobre el estado de ese Terminal, ya que ha llegado al siguiente Terminal y vuelve al estado 3) o una que indique que se ha acabado la información sobre ese estado para todos los terminales (en ese caso pasa al estado 0). En cualquiera de los 2 casos, llama al método creaEstado de Utilidades, que funciona de forma similar al usado en la parte distribuida, ya que el formato de la parte de traza correspondiente a la información de un estado es el mismo en ambas versiones.

A diferencia de la versión distribuida, en esta versión el estado no puede guardarse únicamente en el vector de estados, puesto que luego no sabríamos a que Terminal corresponde dicho estado.

Para evitar esto, diseñamos una nueva estructura, Terminal, en la que guardamos el vector de estados de ese Terminal. Además, llevamos un número entero para indicar la posición que estamos mostrando en ese momento, y otro número entero para indicar el número de estados totales que tenemos en ese Terminal. Además creamos en esta clase un método que nos devuelve el frame con la representación gráfica del estado para que con una simple llamada podamos mostrar directamente el Terminal.

Con esto ya tendríamos guardada la información sobre un Terminal, pero tenemos que guardar la de varios, por lo que creamos otra clase, Terminales, donde llevamos la lista de elementos de la clase Terminal, es decir, los diferentes vectores de estados de los procesos que se están ejecutando simultáneamente en la instancia del programa de Maude que se esta ejecutando.

Los métodos de la clase Terminal nos permiten gestionar la información sobre los terminales, es decir, añadir estados a los terminales, obtener los jframe que queramos, etc. Además, creamos aquí el método que nos permite saber el número de estados creados que llevamos, que será el mínimo entre los números de estados de los

terminales, ya que por ejemplo hasta que el estado 3 no este creado en todos los terminales, es como si no estuviera creado en ninguno, puesto que no podríamos mostrarlo para todos, y decidimos que en esta versión se mostraría la información de todos los terminales al mismo tiempo.

Al igual que en la versión centralizada en esta versión también es necesario ejecutar “cont 1” cuando queremos pasar al siguiente estado de la ejecución de Maude para en el nuevo estado analizar la traza de forma idéntica a la explicada anteriormente en este apartado.

9.4. Vector de Terminales

Ya que en esta nueva versión se deben mostrar simultáneamente los distintos procesos que componen la instancia del programa decidimos que lo más apropiado sería crear la estructura de un vector de terminales en el que cada posición del vector correspondiera únicamente a un terminal siendo que cada terminal representaría a un solo proceso.

A lo que nos referimos con terminal es a un vector de estados (flujo de ejecución) de cada uno de los procesos de la instancia del programa.

9.5. Cambio en la interfaz gráfica

Como hemos indicado anteriormente intentamos reutilizar y mantener la mayor parte de la interfaz gráfica de la versión distribuida ya que sería mucho más simple para el usuario aprender a manejar una sola interfaz gráfica a varias, y además podríamos reutilizar el código que generamos para la versión anterior.

En la versión centralizada, en cada paso de la ejecución no tenemos la información de un solo terminal sino la de varios simultáneamente; esto provoca que el diseño del interfaz gráfico de la versión distribuida no sea correcto para el caso de la centralizada; ya que sólo podríamos mostrar la información de un terminal y en este caso es necesaria mostrar simultáneamente la información de todos los terminales que componen la instancia del programa.

Con la intención de solucionar este problema diseñamos un nuevo interfaz gráfico reutilizando el anterior.

Así podríamos reutilizar la parte externa de la interfaz gráfica (cargar Maude, cargar Programa, cargar instancia . .) y la parte más interna (mostrar un MobileObject, mostrar un ClientRootObject, colorear los atributos que cambian con respecto al estado anterior . .).

Ahora con esta nueva interfaz mostramos varias ventanas dentro de una ventana principal. La ventana principal es parecida a la de la versión distribuida (reutilizamos el menú, los botones de avanzar y retroceder, y la barra de estado).

En el lugar donde antes aparecía la información de un estado (objetos y mensajes) se crean ahora tantas ventanas como procesos existan en la ejecución de la instancia del programa.

Cada una de estas ventanas tiene como título el nombre del proceso, y en su interior se encuentra la representación de objetos y mensajes de ese proceso en el estado en el que nos encontremos. Esta representación interna es la misma que usábamos en al versión distribuida.

La regla aplicada aparece fuera de estas ventanas internas, en la ventana principal entre los botones de avanzar y retroceder.

9.6. Mejoras en la interfaz

Tras una reunión intermedia en el desarrollo de la aplicación que tuvimos con el tutor, llegamos a las siguientes conclusiones sobre la mejora de la interfaz gráfica en la versión centralizada:

- Dar la posibilidad al usuario de redimensionar las ventanas internas, es decir, las ventanas que muestran gráficamente los mensajes y objetos de cada proceso existente en la instancia del programa que se esta ejecutando.

- Visualización de la regla mediante la cual se crea el nuevo estado entre las flechas de “Anterior” y “Posterior”.

- Dar a las ventanas internas la posibilidad de no redimensionarse a su estado inicial al pasar de un estado a otro en la ejecución del programa.

9.7. Destacar los cambios en atributos con respecto al anterior estado

Debido a la gran utilidad que vimos que tenía en la versión distribuida cambiar de color (a rojo) los atributos de los objetos que se han modificado con respecto al estado anterior, decidimos incluir esa mejora en la versión centralizada; por lo que se comportan igual ambas versiones en el cambio de atributos de los objetos respecto al estado anterior.

9.8. Introducir botón de ver traza

Decidimos incluir también en esta versión un botón con el cuál pudiéramos ver exactamente la traza que saca Maude para cada estado.

Ahora, como en la versión centralizada se comprenden todos los procesos que componen la instancia del programa en la misma interfaz, disponemos de un botón “Ver traza” para cada proceso; de forma que en cada estado podemos observar a partir de los distintos botones “Ver traza” la porción de traza correspondiente a cada proceso en el estado que nos encontremos observando en ese momento.

Así podemos observar que la ejecución del programa es la correcta y visualizar partes de la traza que no se observan gráficamente en la aplicación debido a su inferior relevancia.

9.9. Reunión de seguimiento

Nos reunimos con el tutor del proyecto para evaluar una serie de mejoras sobre esta versión de la aplicación. Concluimos que debíamos de dar al usuario las mismas mejoras en la versión centralizada que en la distribuida como por ejemplo la posibilidad de elegir las reglas a partir de las cuales se crearían nuevos estados.

También concluimos que la aplicación debía cargar automáticamente la versión centralizada de Mobile Maude así como mostrar directamente el estado uno al ejecutar cualquier instancia de cualquier programa para facilitar al usuario el uso de la aplicación.

9.10. Archivo de propiedades. Configuración de la ruta de Maude

Como para la versión centralizada es necesario cargar una versión de Mobile Maude distinta tenemos que crear otra propiedad, modificando el archivo de propiedades, para que el usuario pueda especificar donde tiene instalada la versión centralizada de Mobile Maude.

9.11. Elección de las reglas que se van a mostrar

Como en la versión distribuida esta fue una de las mejoras que hacían más fácil de usar y completa nuestra aplicación decidimos reutilizarla.

Gracias a un menú compuesto por “Check boxes” podemos decidir en cualquier momento de la ejecución del programa las reglas que nos llevarán a estados que SI mostraremos gráficamente en la aplicación.

Por ejemplo, podríamos visualizar primeramente todos los estados a los cuales se llega con la regla “do-somehting” para una vez vistos todos estos estados y sin reiniciar la aplicación, añadir una nueva regla de las disponibles en el menú y visualizar tanto los estado a los que se llega con la regla “do-something” como los estados a los que se llega tras aplicar esta nueva regla.

9.12. Muestra del estado 1 del vector automáticamente

Como en las reuniones de seguimiento llegamos a la conclusión de dar al usuario las mismas facilidades de ejecución en la versión centralizada que en la distribuida, esto implicaba añadir también la propiedad de mostrar el estado uno del vector automáticamente. Ya que habíamos realizado la investigación sobre como realizar esto en la versión distribuida aquí nos resultó bastante fácil añadirlo ya que pudimos reutilizar esta parte del código por completo.

9.13. Reunión de seguimiento

Antes de empezar con la tercera parte del proyecto (Model Checker) tuvimos una última reunión acerca de la versión centralizada de la aplicación. En esta reunión decidimos añadir también a la versión centralizada la característica de control de errores y dejar cerrado el desarrollo de esta versión con la última parte, la parte de prueba y error.

9.14. Control de errores

Al igual que en la versión distribuida realizamos sobre la aplicación una estructura “Try and Catch” para capturar las excepciones que pudieran darse como comportamiento anómalo de la aplicación y su posterior estudio por parte de los desarrolladores de la aplicación al dejar registrada la excepción capturada en el log.

9.15. Pruebas y arreglo de errores

En esta última fase, pero no por ello menos extensa, realizamos las distintas pruebas sobre los ejemplos proporcionados por el tutor; para, al igual que en la versión distribuida, poder capturar a través de los fallos producidos en el análisis de la traza de dichos ejemplos las peculiaridades de la traza de Maude; pero esta vez, de la versión centralizada.

10. MODEL CHECKER

10.1. Reunión de seguimiento

En la reunión de seguimiento que tuvimos con el tutor sobre como hacer esta versión de la aplicación nos dieron un par de ejemplos uno con resultado cierto y otro con resultado falso, siendo estos los únicos dos resultados posibles; para poder analizar la traza en ambos casos para desarrollar la aplicación y además probar nuestra versión cuando fuera desarrollada.

10.2. Generación de otro proyecto reutilizando código

En esta tercera y última versión de nuestra aplicación reutilizamos todo el código de la versión centralizada salvo los títulos de la interfaz gráfica, que han sido adaptados a esta nueva versión, y el intérprete y la representación de estados que aunque prácticamente se ha reutilizado todo el código del intérprete de la centralizada, se necesita hacer una modificación ya que en caso de que el resultado de ejecutar ModelChecker sobre una instancia de un programa de Maude sea cierto no habría que analizar más (por ejemplo la parte de analizar si es cierto hay que añadirla al intérprete); pero en caso de que fuera falso habría que analizar el contraejemplo de la misma forma que se analiza la traza resultado de la ejecución de la instancia de un programa de la versión centralizada de un programa Maude.

10.3. Cambios en el intérprete

En este caso partimos del intérprete de la versión centralizada ya que, como hemos explicado anteriormente no había que hacer muchas modificaciones para desarrollar el intérprete de la versión del ModelChecker a partir de la versión centralizada.

Ahora, en esta versión, la traza que muestra Maude, para un resultado cierto sería la siguiente:

información no necesaria

result...true

información no necesaria

Mientras que la traza que muestra Maude, para un resultado falso sería la siguiente:

información no necesaria

result...counterexample

información no necesaria

< pid(0) : . . .

Información sobre el estado, sigue el mismo formato que la información sobre un estado de la versión centralizada y por lo tanto de la distribuida.

mensajes

<objeto>

mensajes

<objeto>

...

<objeto>

mensajes

...

>

...regla aplicada...

< pid (1) : . . .

y así hasta que finalizara el último proceso.

En el nuevo autómata, el estado 0 va pasando línea a línea hasta que se encuentra una línea con la palabra result, lo que indica que hay se encuentra el resultado de aplicar el ModelChecker a la instancia elegida. En ese momento, se actualiza la interfaz con el resultado del ModelChecker y se pasa al estado 0 otra vez. Aquí pueden pasar dos cosas:

- Si el resultado fue cierto se seguirán consumiendo líneas hasta que finalice la traza de Maude y no se mostrará nada más en pantalla.

- Si el resultado fue falso se analiza el contraejemplo; esto quiere decir que al consumir líneas el estado 0 (en el que nos encontramos) si una de las líneas consumidas empieza con "< socketManager" nos encontramos ante un estado del contraejemplo lo que haría pasar al estado 3 y hay una vez encontremos la línea que empieza por "pid(x)" pasar al estado 1 para analizar el proceso, como en el intérprete del la versión centralizada. Una

vez lleguemos de nuevo a una línea que empieza con “pid (x+1)” pasaríamos al estado 3 y así sucesivamente; con lo que por lo que vemos el interprete es muy similar al intérprete de la versión centralizada.

10.4. Cambios en la interfaz gráfica

La interfaz gráfica esta prácticamente reutilizada de la versión centralizada salvo que a la hora de empezar la ejecución de la instancia; se muestra el resultado del modelChecker (cierto o falso) y en caso de que fuera falso se muestra el contraejemplo de la misma forma que se muestra una ejecución de una instancia en la versión centralizada.

10.5. Modificación en la forma de capturar reglas

Como hemos explicado en el punto anterior en el ModelChecker cambia la situación de las reglas ya que ahora no aparece result en cada estado como en la versión centralizada sino que analizamos los nuevos estados a partir de líneas que empiezan con “< socketManager”, situándose ahora las reglas al final de cada estado de la ejecución teniendo que acomodar a esto el interprete de la versión centralizada, ya que como hemos dicho le hemos reutilizado, para esta versión, el ModelChecker.

10.6. Control de errores

Al igual que en las dos versiones anteriores de la aplicación realizamos sobre esta versión una estructura “Try and Catch” para capturar las excepciones que pudieran darse como comportamiento anómalo de la aplicación y su posterior estudio por parte de los desarrolladores de la aplicación al dejar registrada la excepción capturada en el log.

10.7. Prueba y error

Esta última fase no fue muy extensa en esta versión de la aplicación ya que solo disponemos de dos ejemplos uno con resultado cierto y otro falso. Una vez probamos los ejemplos y a partir de los fallos y de las pecualiaridades de las trazas dependiendo del resultado completamos el funcionamiento del interprete reutilizado de la versión centralizada dejamos cerrado el desarrollo de la aplicación.

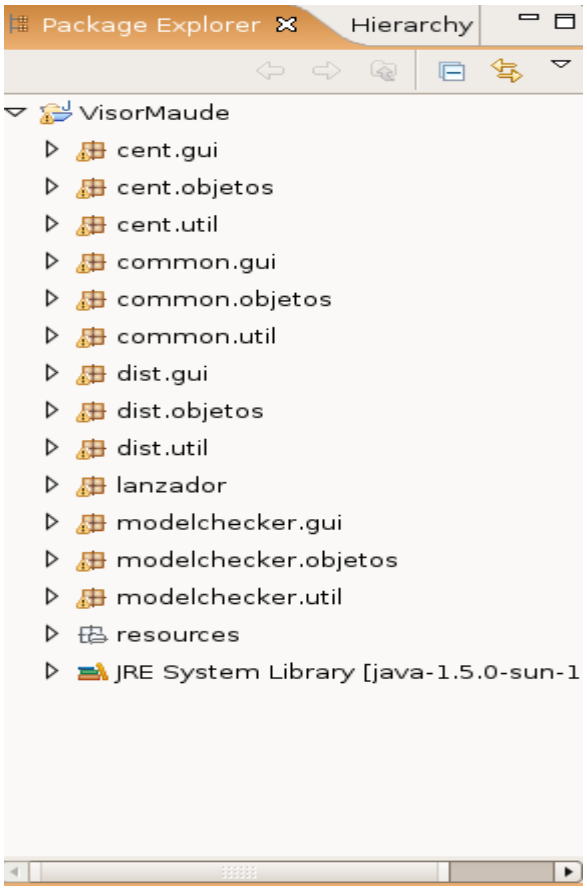
11. UNIÓN

11.1. Reunión de seguimiento

Tuvimos una última reunión de seguimiento para hablar sobre la entrega final del proyecto. En un principio teníamos tres aplicación distintas (distribuida, centralizada y model checker) y como vimos que las tres aplicaciones eran muy similares decidimos unirlas en una sola, para así facilitar su uso y análisis. Así si en un futuro alguien quisiera continuar mejorando el desarrollo le sería más fácil. Para ello decidimos crear una interfaz común, rehusar código y reestructurarlo en paquetes.

11.2. Reestructuración del código

Para optimizar el código pensamos en reunificar el código generando una estructura de paquetes en la que diferenciamos entre las partes específicas. Así quitamos las partes del código que se repetían para cada versión y las juntamos sólo en una.



Como vemos en la imagen los paquetes se separan por aplicación y a la vez por utilidades. Es decir, para cada aplicación (centralizada, distribuida y model checker) y para la parte común tenemos un paquete para la interfaz gráfica (gui), otro para utilidades, por ejemplo para el análisis de la traza (util) y otra para la representación de los datos de los objetos (objetos). Luego aparte de estos paquetes por aplicación tenemos el paquete que contiene la clase principal (lanzador).

11.3. Creación de una nueva ventana inicial del proyecto

El siguiente paso era hacer una interfaz común que permitiera poder elegir una u otra aplicación. Para ello también tuvimos que hacer algunos cambios en el código pero tampoco muy complicados. En la versión final cuando se ejecuta el proyecto sale una interfaz con tres botones (uno para cada aplicación). Según el botón que pulsemos saldrá la pantalla correspondiente a la aplicación (la mismas que cuando las aplicaciones estaban separadas).

11.4. Optimización del uso del vector de traza

Para evitar el exceso de consumo de memoria del PC hemos optimizado el uso del vector de traza. Antes se iba recorriendo el vector y ahora el vector de traza se usa como buffer, el hilo lector mete el código y el intérprete va analizando, con la diferencia de que en vez de irse moviendo por el vector como hacia antes, ahora va consumiendo posiciones del vector y eliminándolas.

El funcionamiento del vector se basa en que como el vector funciona como un buffer, llega a la posición 5, pero cuando esta en la posición 5 ya no va avanzando la posición a estudiar, sino que elimina la primera, y así el efecto es el mismo, pero conseguimos que el vector no crezca infinitamente. Se hace en la posición 5 porque hay que guardar unas cuantas líneas ya que a veces hay que mirar lo que ponía líneas atrás (por ejemplo para saber la regla).

11.5. Implementación de la opción de resetear sistema

Se finalizan los hilos lectores e intérprete y se reinician los procesos y las estructuras que contienen la información (vector de estados en la distribuida y terminales en la centralizada y model-checker). Para ello se usa el semáforo, haciendo que cada hilo vaya saliendo del bucle infinito en el que esta. Una vez que todos los hilos han acabado, destruye los procesos y los crea de nuevo (carga maude de nuevo).

11.6. Prueba y error

En esta última fase, pero una de las más importante, seguimos realizamos pruebas para cerciorarnos de que no hay ningún fallo que no hayamos visto anteriormente, dejando una versión a punto para la entrega final.

11.7. Comentarios del código y JavaDoc

Una vez que ya tenemos una versión definitiva, el último paso era acabar de comentar el código para así hacer más fácil su entendimiento y facilitar su uso en el caso de que otras personas retomaran el proyecto para cualquier otra utilidad.

Tras comentar el código usamos una utilidad del Eclipse que permite generar JavaDoc, que es la información estructurada de la misma forma que en la página de Java, la cual anexaremos al final de la memoria para su lectura.

12. IMPLEMENTACIÓN

12.1. Estructura de carpetas

Hemos entregado un .zip llamado SSII.zip. Al descomprimir la carpeta vemos que contiene las siguientes carpetas:

- **VisorMaude**: carpeta que contiene el proyecto para poder ser abierto en eclipse. Contiene tanto los ficheros .class como los .java. Están organizados por paquetes:
  - **cent**: contiene las clases referentes a la parte de la versión centralizada.
  - **common**: contiene las clases comunes a las tres versiones.
  - **dist:** contiene las clases referentes a la parte de la versión distribuida.
  - **lanzador:** contiene la clase principal (starter).
  - **modelchecker:** contiene las clases referentes al modelchecker.
  - **resources:** carpeta que contiene las imágenes que se usan en la aplicación(flechas de anterior y siguiente).

- **CodigoMaude:** que contiene Maude, las versiones centralizada y distribuida de Mobile Maude y los ejemplos de cada una de las versiones**:**
  - **Maude-linux:** contiene Maude 2.3.
  - **Mobile-Maude:** contiene la versión distribuida de Mobile Maude y los ejemplos correspondientes.
  - **Centralized-mm:** contiene la versión centralizada de Mobile Maude y los ejemplos para la versión centralizada y el model checker.

- **Ejecución:** contiene el archivo de propiedades (en los que se encuentran las rutas de donde se encuentra Maude y las versiones distribuida y centralizada de Mobile Maude), el archivo para el logger, el archivo proyecto.sh (el cual contiene la ruta de la máquina virtual de java, el .jar a ejecutar y el archivo de propiedades).

12.2. Pasos para la ejecución de la aplicación

En la carpeta incluimos tanto un archivo .jar (proyecto.jar) que se ejecuta a partir del .sh (proyecto.sh) y el proyecto para abrirlo con eclipse (carpeta VisorMaude).

12.2.1. Ejecutar .Jar

1º. Lo primero que necesitamos es cambiar en el archivo proyecto.sh de la carpeta ejecución la ruta de Java.

2º. El siguiente paso es cambiar las rutas de Maude y Mobile Maude para la versión distribuida y centralizada en el archivo config.properties. También tenemos que cambiar la ruta del directorio donde se encuentra el archivo de propiedades para el logger.

3º. Después lo que tenemos que cambiar es la ruta de la salida del logger en el archivo logger.properties.

4º. Tras los pasos anteriores ya solo queda ejecutar el archivo proyecto.sh en un terminal.

12.2.2. Ejecutar proyecto en Eclipse

1º. Lo primero que necesitamos es cambiar las rutas de Maude y Mobile Maude para la versión distribuida y centralizada en el archivo config.properties. También tenemos que cambiar la ruta del directorio donde se encuentra el archivo de propiedades para el logger.

2º. El siguiente paso es cambiar la ruta de la salida del logger en el archivo logger.properties.

3º. Una vez que hemos cambiado las rutas, abrimos el eclipse. Elegimos como worskpsace la carpeta SSII. Como se ve se muestra la carpeta VisorMaude que es la que contiene el código a ejecutar.

4º. Para poder ejecutar la clase Starter.java (es la que contiene la clase principal) tenemos que configurar el eclipse. Para ello vamos al menú Run/run en Main elegimos la clase principal (Starter) y en arguments dentro de program arguments ponemos la ruta del archivo config.properties.

13. ESTRUCTURACIÓN DEL CÓDIGO

Aunque al principio teníamos un proyecto para cada aplicación, finalmente decidimos que lo mejor era unirlos todos en uno, debido a las características similares que tenían.

Hay muchas clases que son parecidas para las distintas versiones de la aplicación. Ante la posibilidad de conservarlas como clases distintas o unirlas y en sus métodos ir comparando, hemos elegido la opción de conservarlas como clases distintas. El motivo es que si en algún momento hay que depurar o cambiar algo de alguna de las versiones es así mucho más fácil.

Por lo tanto estructuramos el código en paquetes: common, dist, cent y modelchecker. Cada uno de esos paquetes se divide en subpaquetes: gui, objetos y util. Además tenemos el paquete Lanzador, que simplemente tiene la clase que se encarga de lanzar la aplicación.

13.1. Paquete common

En este paquete están las clases que son comunes a las tres versiones de la aplicación. Se divide en tres subpaquetes:

13.1.1. GUI

Es el paquete correspondiente a las clases del interfaz gráfico. Contiene:

- **InicioDialog:** JFrame de inicio de la aplicación general, que da a elegir cual de las tres aplicaciones quiere ejecutarse.

- **PanelMO**: JPanel que muestra gráficamente la información de un MobileObject. Es común puesto que la información que se debe mostrar sobre un MobileObject es la misma en las tres aplicaciones.

- **PanelRO:** JPanel que muestra gráficamente la información de un RootObject. Es común puesto que la información que se debe mostrar sobre un RootObject es la misma en las tres aplicaciones.

- **StatusBar:** Jpanel que contiene la barra de estado, que irá colocada en la parte inferior de la pantalla y que utilizamos para mostrar el número de estados, el estado en que se encuentra el usuario y el estado de la aplicación (cargando, etc..).

13.1.2. OBJETOS

En este paquete tenemos las clases que nos sirven para representar los datos de los objetos. Contiene:

- **Mensaje:** Objeto que usamos para guardar la información de un mensaje. Tiene un atributo numérico que indica el tipo de mensaje que es, un booleano para indicar si el mensaje es o no relevante, y un vector para guardar los distintos atributos que tenga el mensaje, que son distintos dependiendo del tipo de mensaje.

- **MobileObject:** Objeto que usamos para guardar la información de un MobileObject. Los datos relevantes de un MobileObject son el nombre, el valor del atributo S, el valor del atributo GAS y el valor del atributo HOPS, por lo que esta clase tiene esos atributos.

- **RootObject:** Objeto que usamos para guardar la información de un RootObject. Los datos relevantes de un RootObject son el nombre, el valor del atributo CNT, el del atributo GUESTS y el del atributo FORWARD, por lo que esta clase tiene esos atributos.

- **ObjetoMostrable**: Clase padre de MobileObject y de RootObject, y que sirve para facilitarnos la gestión de los objetos en el intérprete, al no tener que estar siempre distinguiendo entre los dos tipos de objetos.

13.1.3. UTIL

En este paquete tenemos las clases de utilidad, que se encargan del análisis de la traza. Contiene:

- **Códigos:** Clase en la que guardamos las constantes que nos sirven para parametrizar los valores, por ejemplo el nombre de los atributos en la traza. Así, si la traza de Maude cambia y aparecen nuevos nombres de atributos, solo tendríamos que cambiar aquí su valor, y no en todas las posibles apariciones del String en el código fuente

- **HiloLector:** Hilo encargado de capturar la traza e ir guardándola en un vector que se le pasa por parámetro cuando se le construye. Este hilo es común puesto que no trata de ninguna forma la traza, simplemente la guarda en el vector.

- **Semáforo:** Clase que se usa como semáforo para poder parar los hilos cuando se necesite. Simplemente tiene unos bloqueos para evitar problemas de acceso concurrente, y un booleano, que servirá para que los hilos lo comparen dentro de su bucle global y sepan cuando deben salir de él.

13.2. Paquete dist

En este paquete incluimos las clases que son necesarias para la aplicación distribuida. Este paquete se divide en otros tres subpaquetes:

13.2.1. GUI

Es el paquete correspondiente a las clases del interfaz gráfico. Contiene:

- **DialogoTraza:** Es el JFrame que usamos para mostrar la traza de un estado en modo texto. Se compone únicamente de un textarea. No es común puesto que en este caso el comportamiento es distinto que en la versión distribuida y que en el model-checker, ya que aquí bloqueamos el frame principal hasta que esta ventana se cierra.

- **PantPrincipal:** Es el JFrame principal de la aplicación. Contiene un menú, un par de botones para avanzar y retroceder y una barra de estado. En el centro de este frame es donde se ira mostrando la

información de los estados, por los que se irá moviendo el usuario mediante los botones de Avanzar y retroceder. La información de los estados se muestra como cajas dibujadas sobre el fondo, en cuyo interior aparecen los nombres de los atributos y sus valores. Estos valores son mostrados en la caja o en una nueva caja que se muestra al colocar el ratón sobre el atributo (esto ocurre cuando el valor es demasiado grande como para mostrarlo en el interior de la caja). Los mensajes del estado aparecen en la parte inferior de la pantalla.

13.2.2. OBJETOS

En este paquete tenemos las clases que nos sirven para representar los datos de los objetos. Contiene:

- **Estado:** En los objetos de esta clase guardamos la información sobre un estado. Se guarda la regla aplicada, un vector de mensajes y otro de objetos, además de un string con la parte de traza correspondiente al estado.

13.2.3. UTIL

En este paquete tenemos las clases de utilidad, que se encargan del análisis de la traza. Contiene:

- **Interprete:** Clase interprete. Esta definida como hilo, para que pueda ejecutarse sin tener que bloquear el flujo principal de la aplicación. Es un bucle que pasa por un método que dependiendo del valor de un entero llama a uno u a otro método. Conseguimos implementar de esta manera el autómata que se encarga de ir analizando la traza.

- **Utilidades:** En el estado 1 del intérprete, se va guardando en un vector la información sobre el estado que se esta analizando de la traza. Este vector se manda al método creaEstado de Utilidades. En la clase Utilidades tenemos por tanto definidos todos los métodos que son necesarios para que partiendo de un vector con la parte de traza correspondiente a un estado, consigamos analizar esa porción de traza y generar la estructura que usamos en nuestra aplicación para guardar la información, es decir, el objeto de la clase estado, los objetos de la clase MobileObject y RootObject, y los mensajes.

- **Union:** Esta clase es la que se encarga de la gestión de la conexión de nuestra aplicación con Maude y Mobile-Maude. Contiene los hilos lectores, los procesos, etc. Es la clase en la que tenemos implementados los métodos que nos permiten enviar instrucciones al proceso de maude para ser ejecutadas.

13.3. Paquete cent

En este paquete incluimos las clases que son necesarias para la aplicación centralizada. Este paquete se divide en otros tres subpaquetes:

13.3.1. GUI

Es el paquete correspondiente a las clases del interfaz gráfico. Contiene:

- **DialogoTraza:** Es el JFrame que usamos para mostrar la traza de un estado en modo texto. Se compone únicamente de un textarea. No es común puesto que en este caso el comportamiento es distinto que en la versión distribuida y que en el model-checker, ya que aquí bloqueamos el frame principal hasta que esta ventana se cierra.

- **PantPrincipal:** Es el JFrame principal de la aplicación. Contiene un menú, un par de botones para avanzar y retroceder y una barra de estado. En el centro de la pantalla aparecerán tantos frames como "terminales" tenga la traza de maude (con terminales nos referimos a lo que maude llama procesos mediante pid(x)). En cada uno de esos frames internos se muestra la información de la misma forma que se hacia en la aplicación distribuida.

13.3.2. OBJETOS

En este paquete tenemos las clases que nos sirven para representar los datos de los objetos. Contiene:

- **Estado:** En los objetos de esta clase guardamos la información sobre un estado. Se guarda la regla aplicada, un vector de mensajes y otro de objetos, además de un string con la parte de traza correspondiente al estado.

- **Terminal:** Cada terminal viene caracterizado por un vector de estados, un número de estados y un entero que indica la posición en la que se está en ese momento.

- **Terminales:** Clase que nos sirve para gestionar los terminales, añadirles proyectos, calcular el número de estados, obtener JPanels con la representación gráfica de un determinado terminal en un determinado estado, etc.

13.3.3. UTIL

En este paquete tenemos las clases de utilidad, que se encargan del análisis de la traza. Contiene:

- **Interprete:** Clase interprete. Esta definida como hilo, para que pueda ejecutarse sin tener que bloquear el flujo principal de la aplicación. Es un bucle que pasa por un método que dependiendo del valor de un entero llama a uno u a otro método. Conseguimos implementar de esta manera el autómata que se encarga de ir analizando la traza.

- **Utilidades:** En la clase Utilidades tenemos por tanto definidos todos los métodos que son necesarios para que partiendo de un vector con la parte de traza correspondiente a un estado, consigamos analizar esa porción de traza y generar la estructura que usamos en nuestra aplicación para guardar la información, es decir, el objeto de la clase estado, los objetos de la clase MobileObject y RootObject, y los mensajes

- **Union:** Esta clase es la que se encarga de la gestión de la conexión de nuestra aplicación con Maude y Mobile-Maude. Contiene los hilos lectores, los procesos, etc. Es la clase en la que tenemos

implementados los métodos que nos permiten enviar instrucciones al proceso de maude para ser ejecutadas.

13.4. Paquete modelchecker

En este paquete incluimos las clases que son necesarias para la aplicación del Model Checker. Son las mismas clases que en la versión centralizada, pero tienen pequeñas variaciones. Esto nos ha hecho decantarnos por la opción de dejarlas separadas en paquetes distintos, puesto que lo consideramos más útil de cara a futuras ampliaciones de la aplicación.

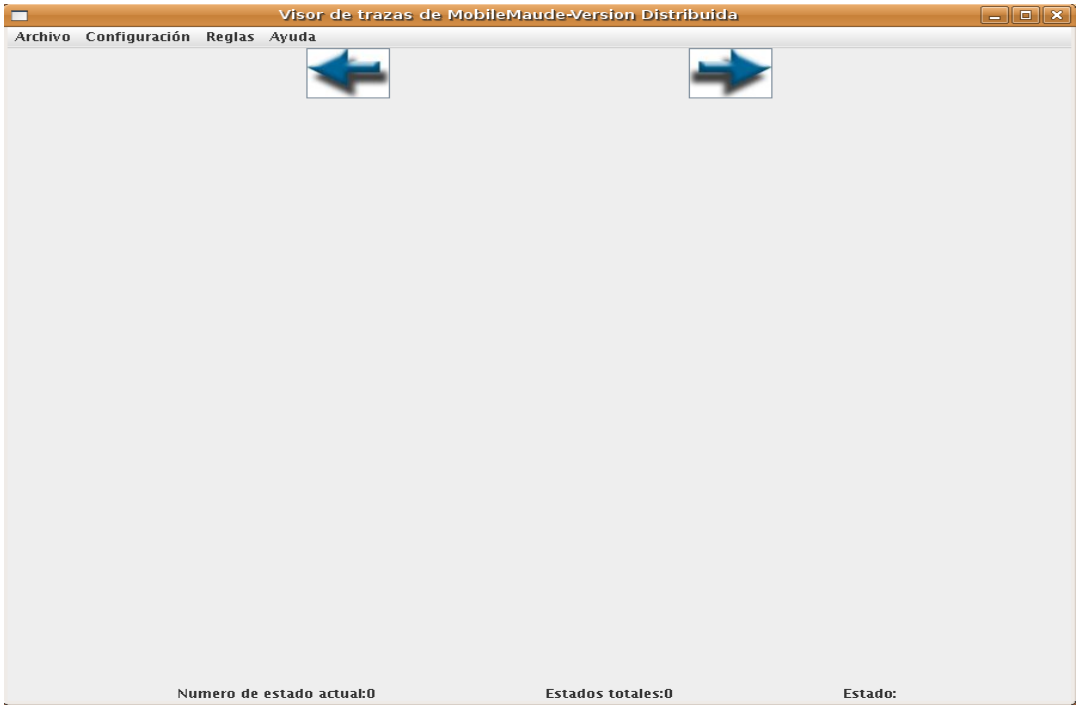
14. FUNCIONAMIENTO Y MANUAL DE USO

Al ejecutar la aplicación nos sale una ventana en la que se nos da la opción de elegir cualquiera de las tres versiones:

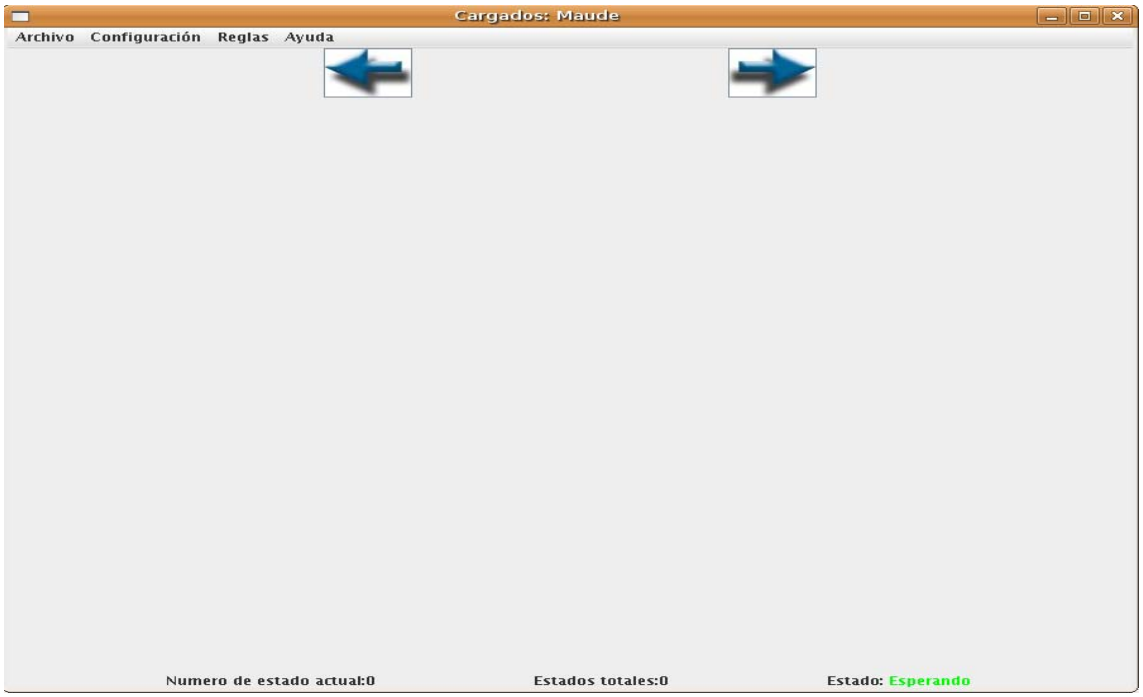
Tras elegir una de las opciones, el funcionamiento varía. A continuación explicaremos mediante pantallas y aclaraciones cómo es el funcionamiento de cada una de las opciones.

14.1. Funcionamiento de la versión distribuida

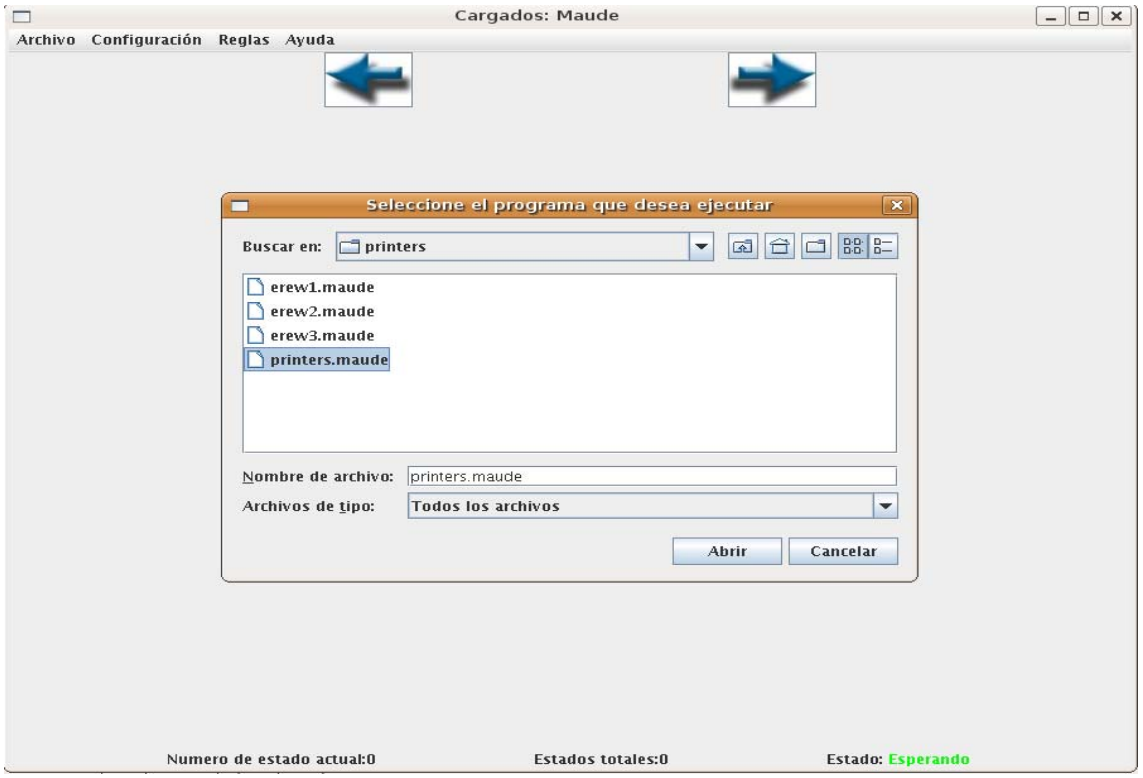
La pantalla principal de la versión distribuida es de la siguiente manera:

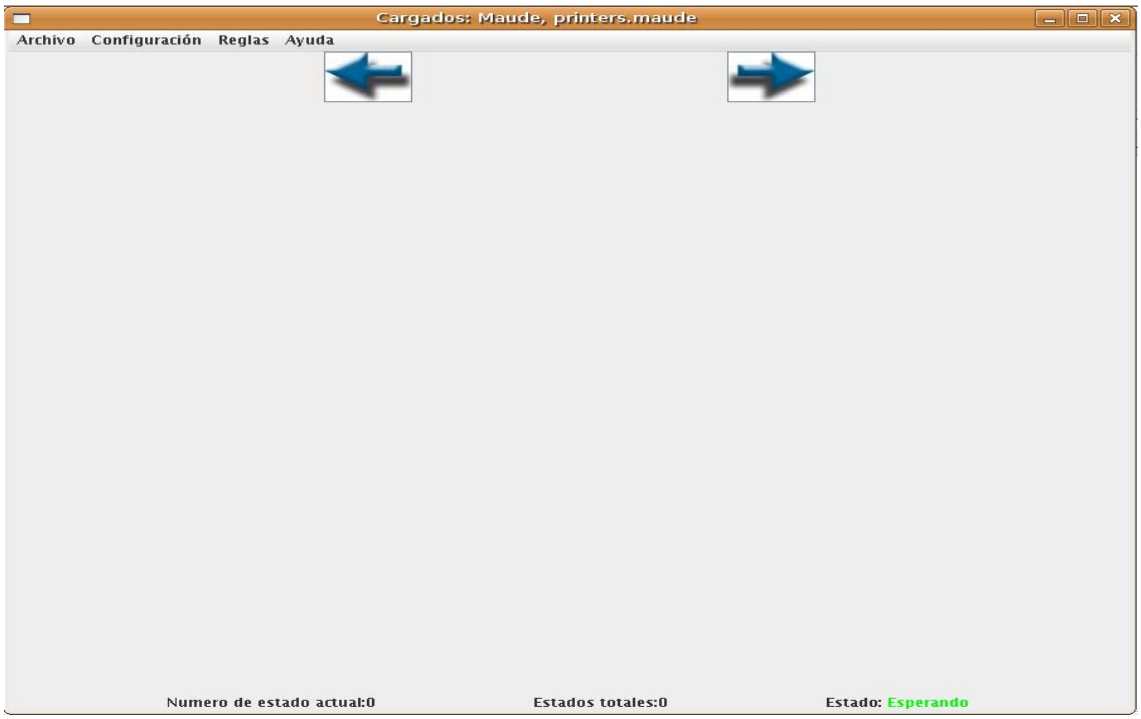


El siguiente paso es cargar Maude. Se puede hacer seleccionando en el menú Archivo la opción cargar Mobile-Maude o mediante el teclado con CTRL+ m.

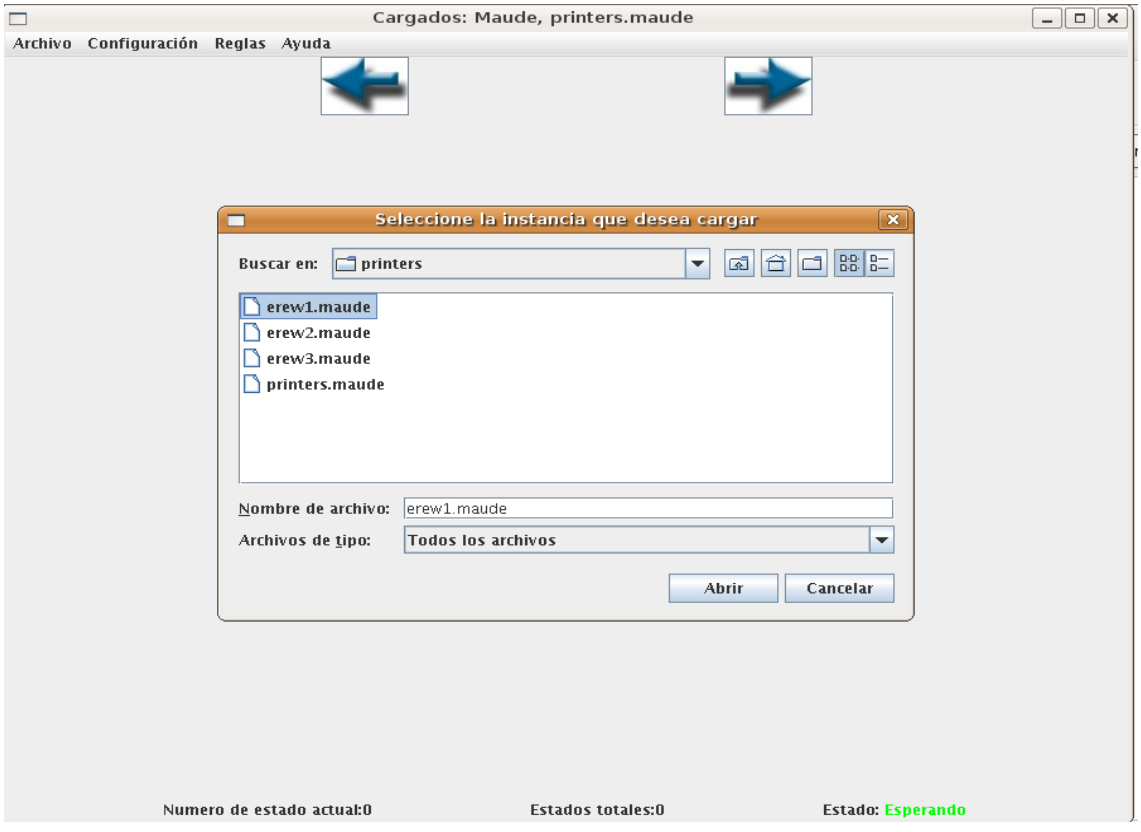


Como vemos arriba ya está cargado Maude, ahora el siguiente paso es cargar el programa (en nuestro caso estamos probando con printers.maude) mediante el menú y eligiendo Cargar Programa o mediante CTRL+a.



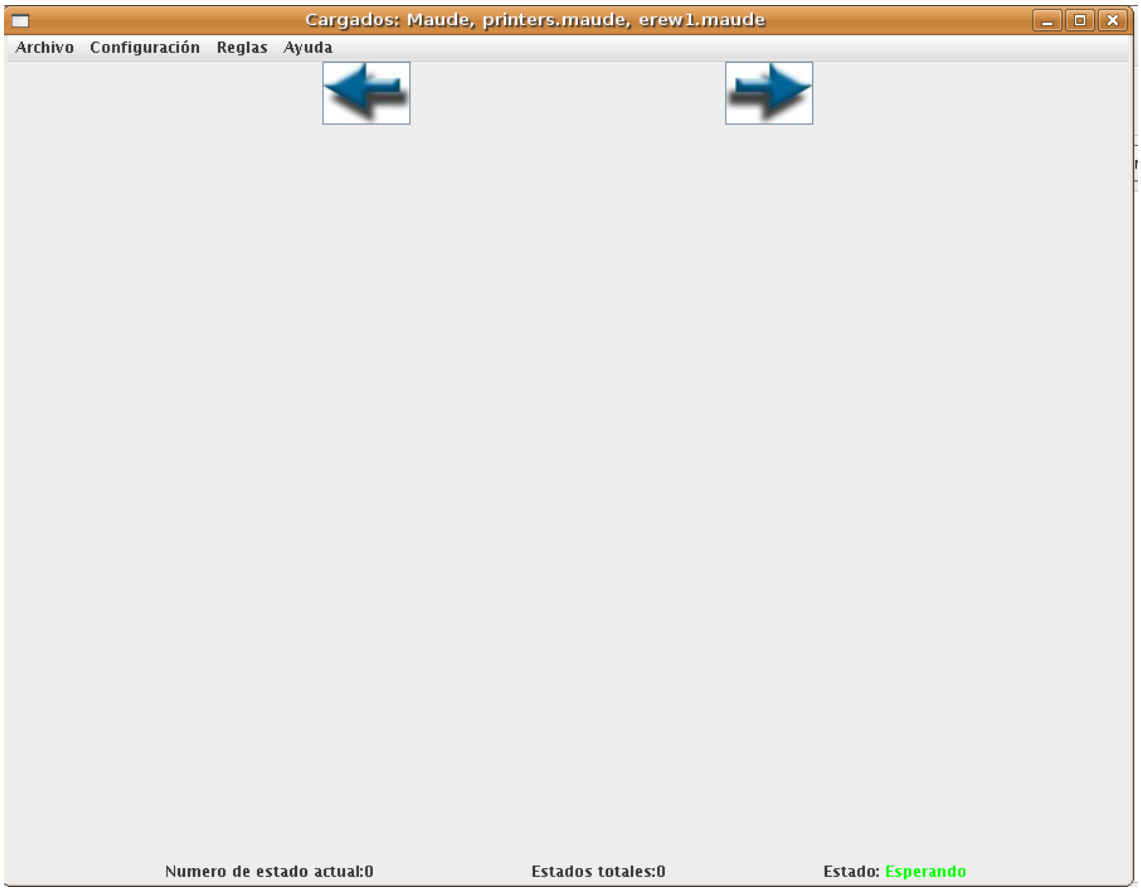


Una vez que hemos cargado el programa, queda cargar la instancia. En este caso cargamos la primera instancia (erew1.maude). Debemos seleccionar del menú Cargar Instancia o pulsar CTRL+ i.



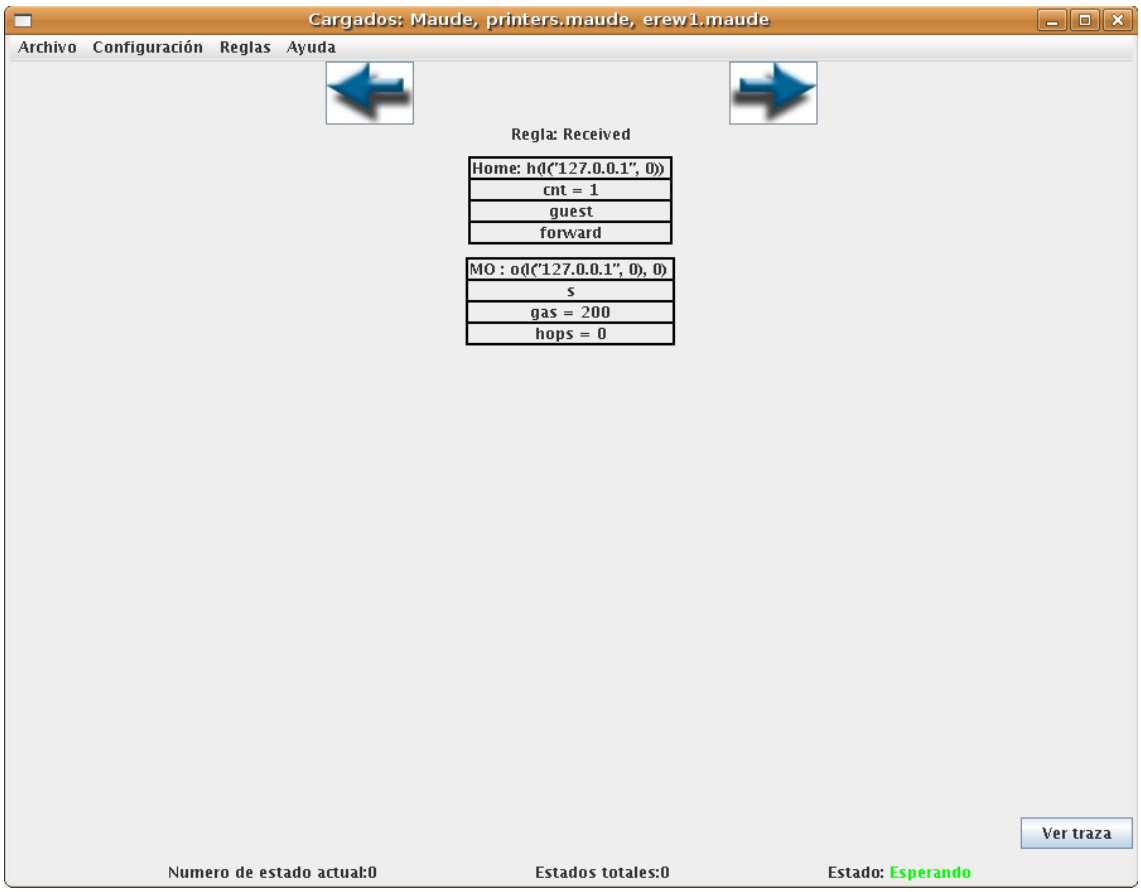
Una vez cargada la instancia, se generarán los estados, actualizándose el número de estados totales, el número de estado actual mostrará el estado en el que nos encontramos, y en estado se mostrará cargando si se están generando estados, y esperando si ha acabado de leer la traza disponible pero espera en caso de que llegue nueva traza. Con las flechas de izquierda y derecha podemos movernos por los distintos estados, desde el primero hasta el último, tanto pulsando la tecla con el ratón como manejándolo con teclado.

Como vemos en el título muestra las tres cosas que hemos cargado (Maude, el programa printers y su instancia erew1). En este caso no tenemos estados, ya que simplemente tendría el estado inicial pero que no lo mostramos.

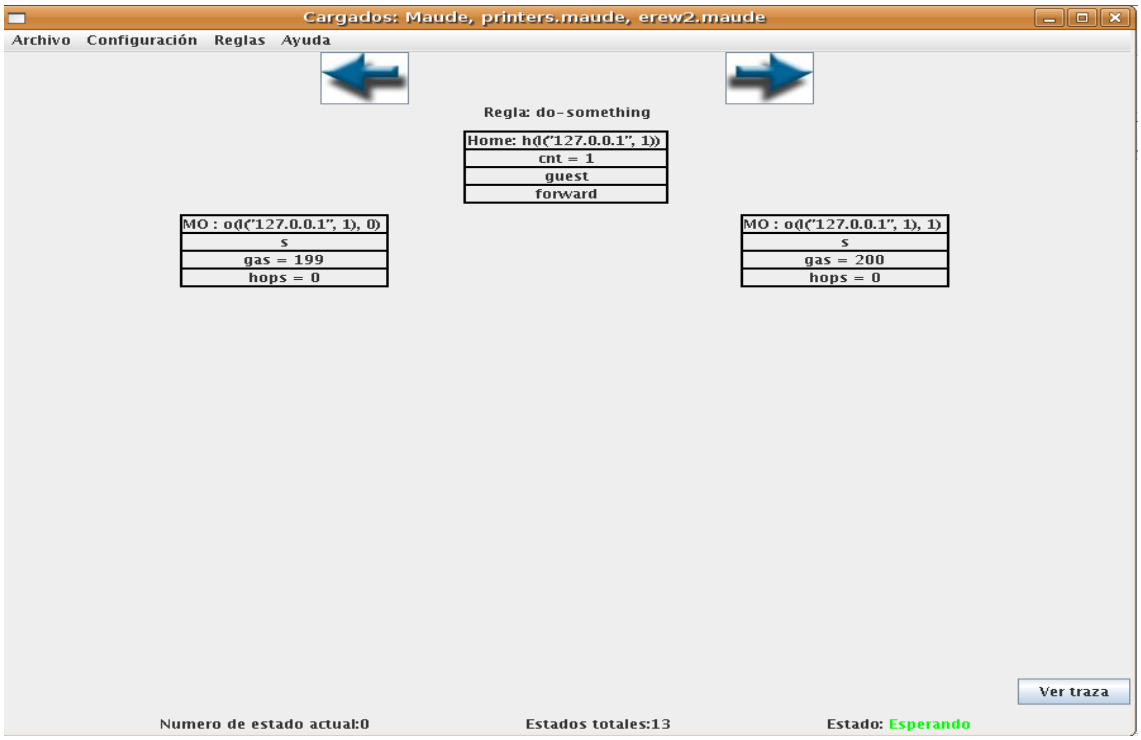


Si queremos cargar otra instancia del mismo programa, deberíamos ejecutar otra aplicación eligiendo la versión distribuida y cargando otra vez Maude, el programa y la nueva instancia (en este caso cargaremos printers.maude y erew2.maude). En este caso se generan estados nuevos tanto en la nueva instancia como en la anterior instancia cargada al comunicarse entre ellas:

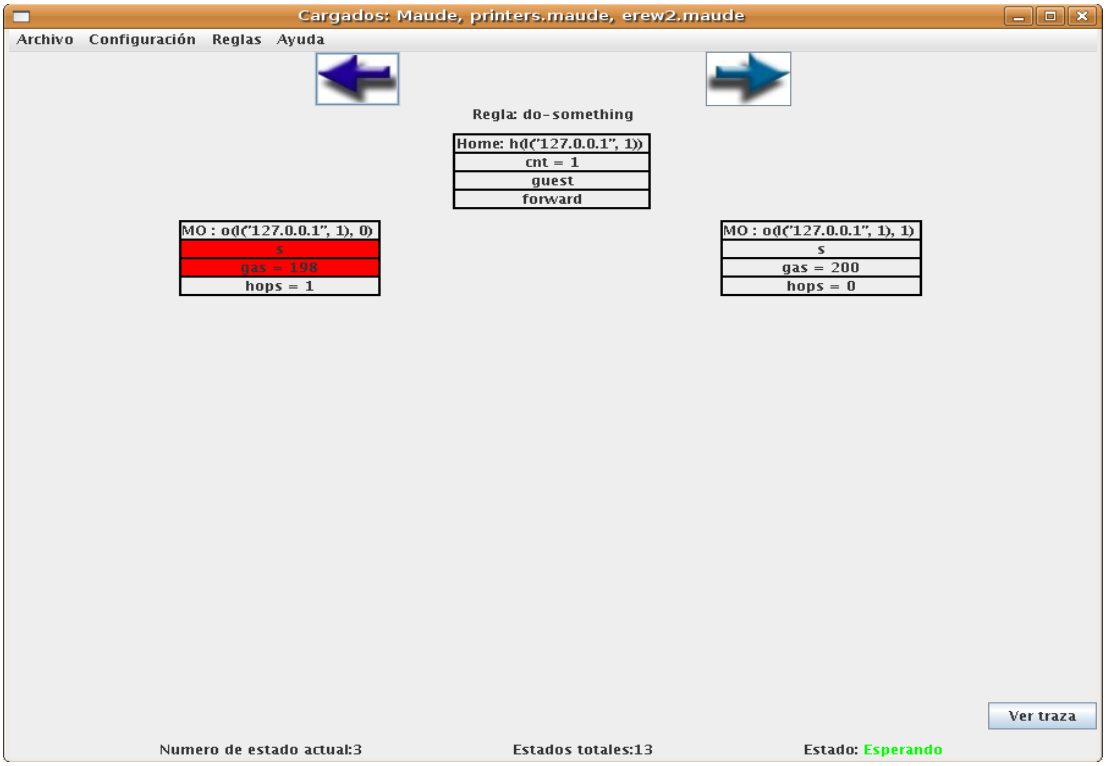
Por lo tanto la instancia 1(erew1) ahora tendrá más estados, en este caso un estado más. Como vemos se muestra la regla del estado y los objetos que tenemos en este estado:



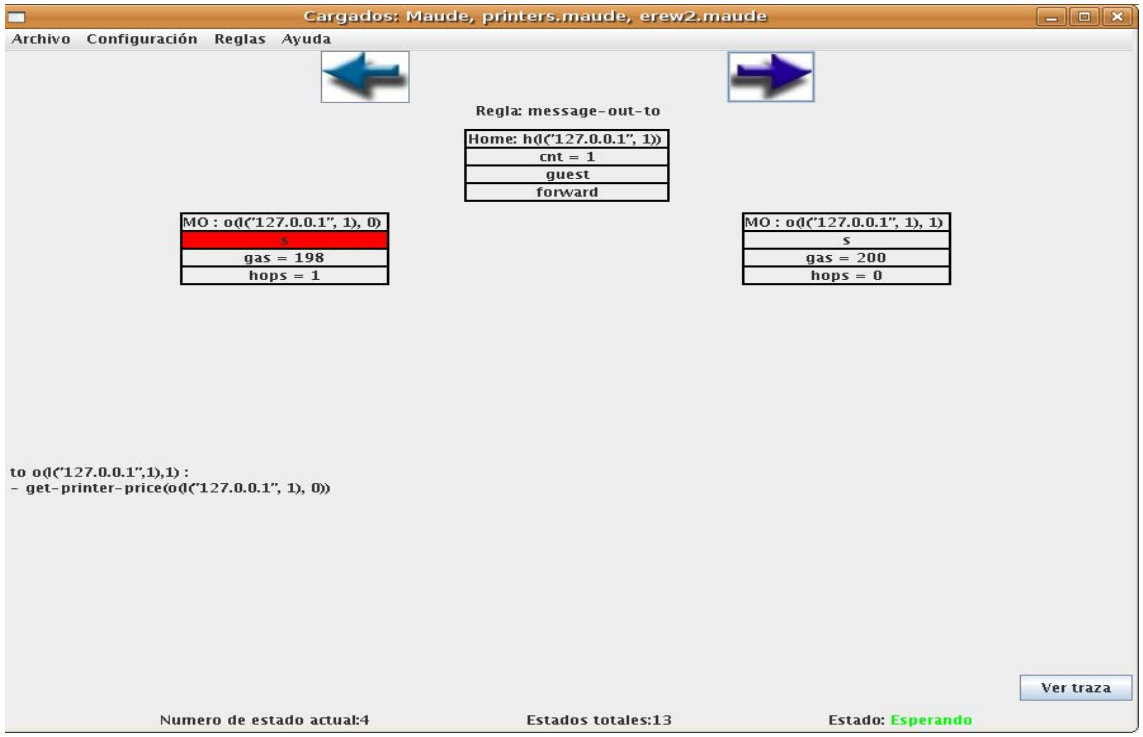
La instancia erew2 como vemos tiene más estados



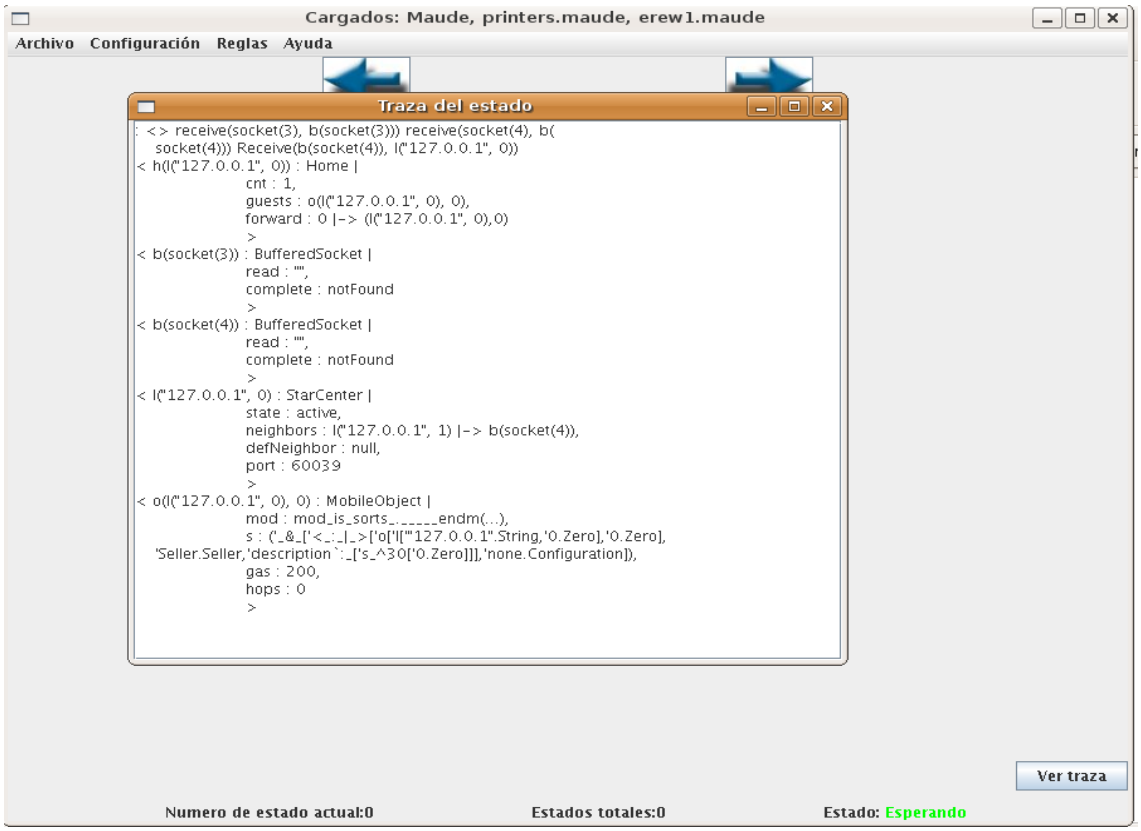
Si nos vamos moviendo por los estados vemos que se va actualizando el número de estado actual y también vemos resaltados en rojo aquellos atributos que en el anterior estado tenían otro valor distinto:



Si hubiera algún mensaje también se mostraría de la siguiente manera:



Como vemos en cada estado tenemos un botón ver traza que si lo pulsamos muestra la traza correspondiente al estado en el que nos encontramos:

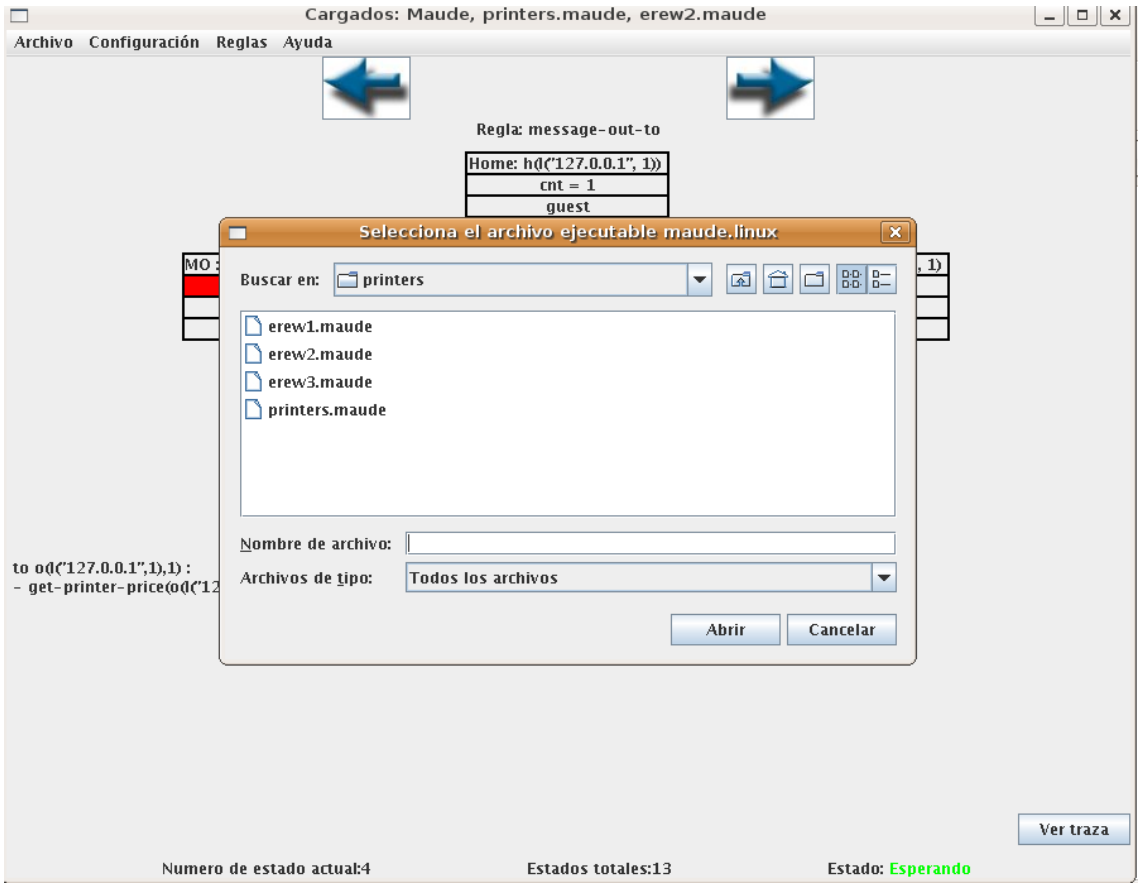


Algunas de las opciones que podemos hacer mediante la aplicación, es Resetear, es decir en una aplicación ya abierta, podemos volver a cargar otro programa y/o instancia diferente. Para ello en el menú archivo elegimos la opción Volver a Cargar Maude o pulsando CTRL +R.

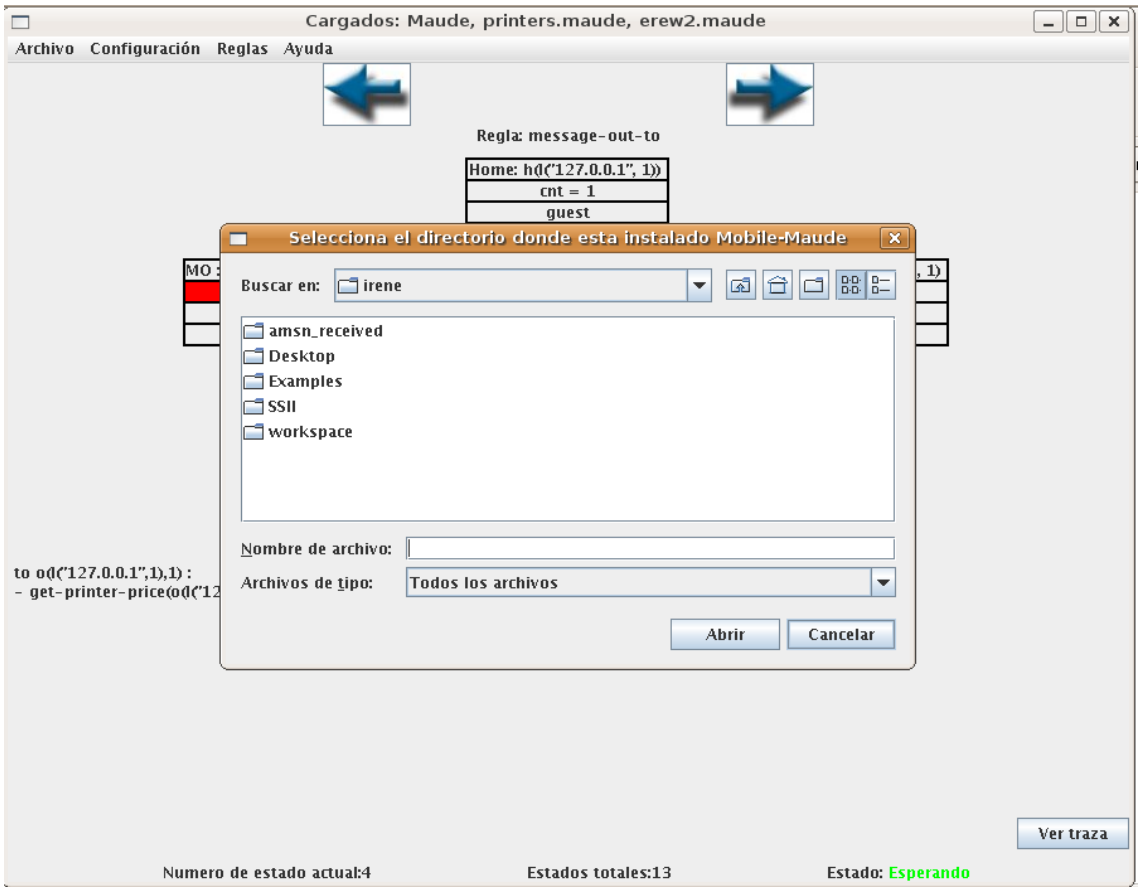
Otra de las opciones que tenemos es poder elegir las reglas que queremos mostrar o no. Para ello en el menú Reglas tenemos una lista de reglas fijas. En principio están todas seleccionadas pero si no queremos mostrar alguna simplemente tenemos que deseleccionarla y todos los estados referentes a esa regla no se mostrarán. Esto se puede cambiar tanto antes de cargar la instancia como una vez que ya esté cargada.

Por último también permitimos cambiar la ruta donde se encuentra cargado el Maude y el Mobile Maude para la versión centralizada en tiempo de ejecución. En este caso no se modifica en el archivo config.properties sino que sólo valdría mientras esa aplicación este abierta.

Para ello si queremos cambiar la ruta del Maude seleccionamos la opción Configuración, luego Ruta Maude y ahí cambiar Ruta. Elegimos la nueva ruta que como hemos explicado en el párrafo anterior sólo es válida para esa aplicación.



Si queremos cambiar la ruta de la versión distribuida de Mobile Maude, tenemos que seguir los mismo pasos pero seleccionando dentro del Menú Configuración la opción Mobile Maude.

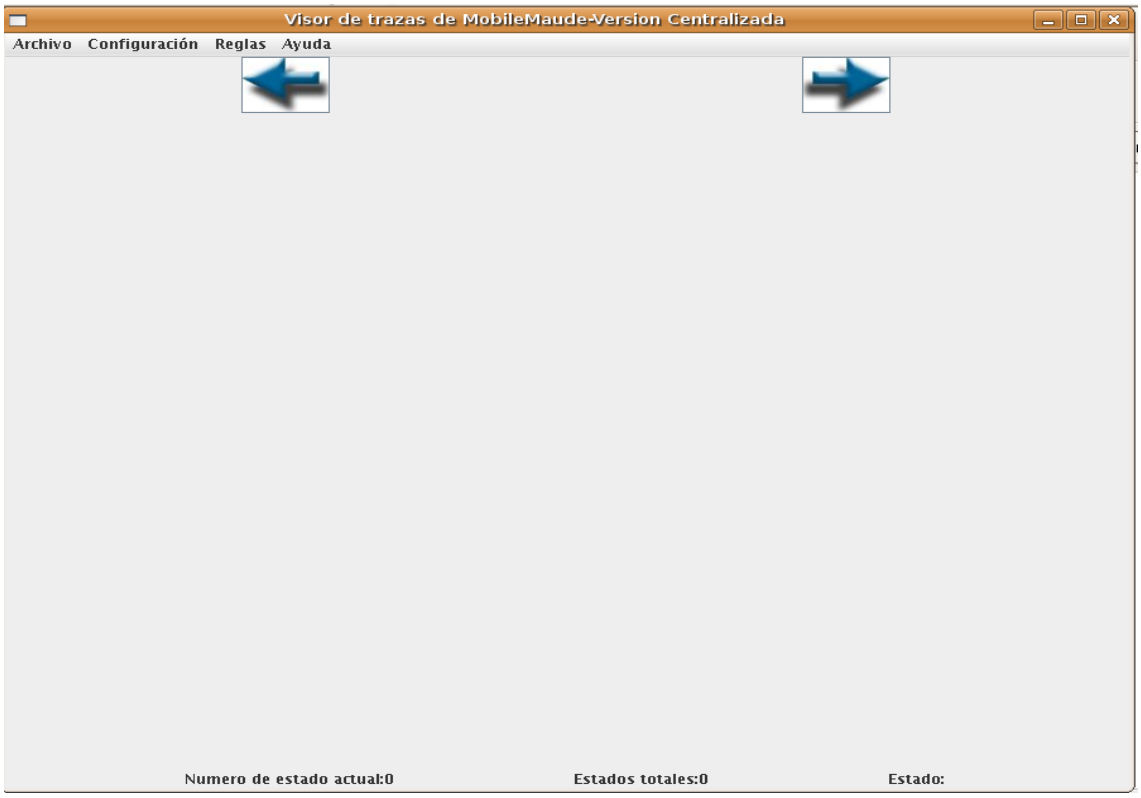


14.2. Funcionamiento de la versión centralizada

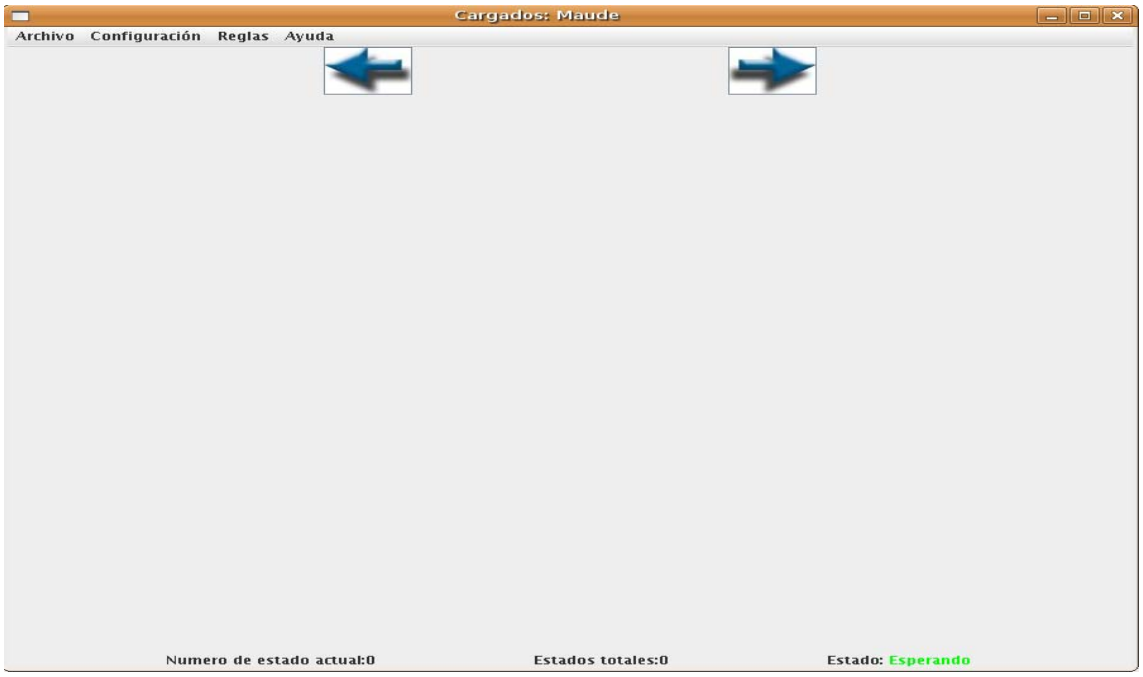
El funcionamiento de la versión centralizada es muy parecido al de la distribuida pero con la principal diferencia de que en la distribuida hay que abrir una aplicación por cada instancia que queremos cargar y en la centralizada de una misma aplicación tenemos todas las instancias de un programa. De hecho en la versión centralizada simplemente tendremos que cargar Maude, Mobile Maude para la versión centralizada y el programa que ya contiene todas las instancias, es decir que sol necesitamos hacer este paso 1 vez para tener todas las instancias.

Para elegir la versión centralizada simplemente basta con elegir esta opción dentro del Menú Principal de la aplicación.

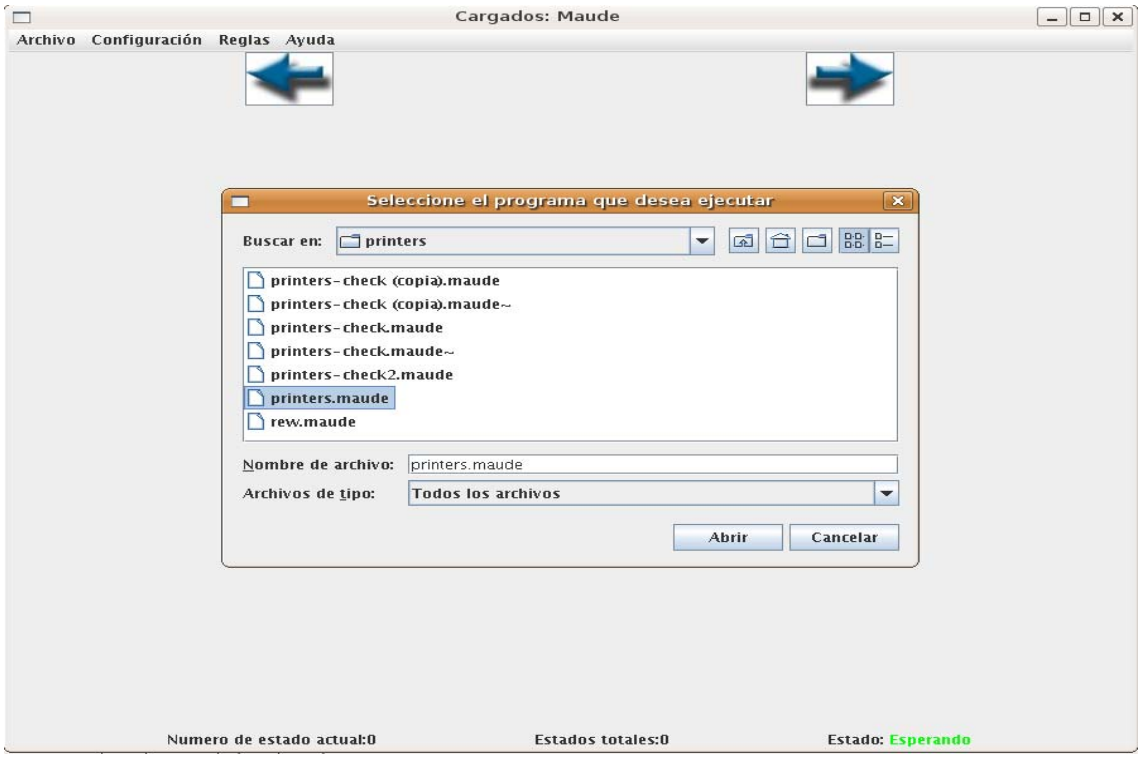
Una vez pulsado el botón, la pantalla principal de la versión centralizada es de la siguiente manera:

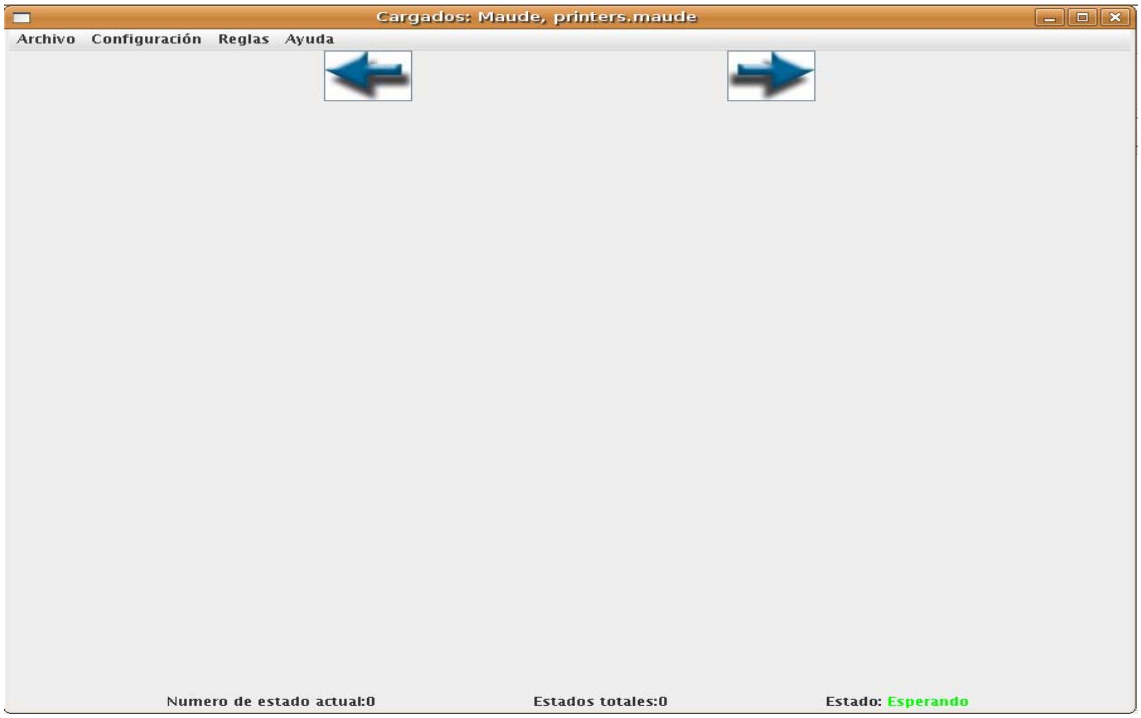


El siguiente paso es cargar Maude. Se puede hacer seleccionando en el menú Archivo la opción Cargar Mobile-Maude o mediante el teclado con CTRL+ m.

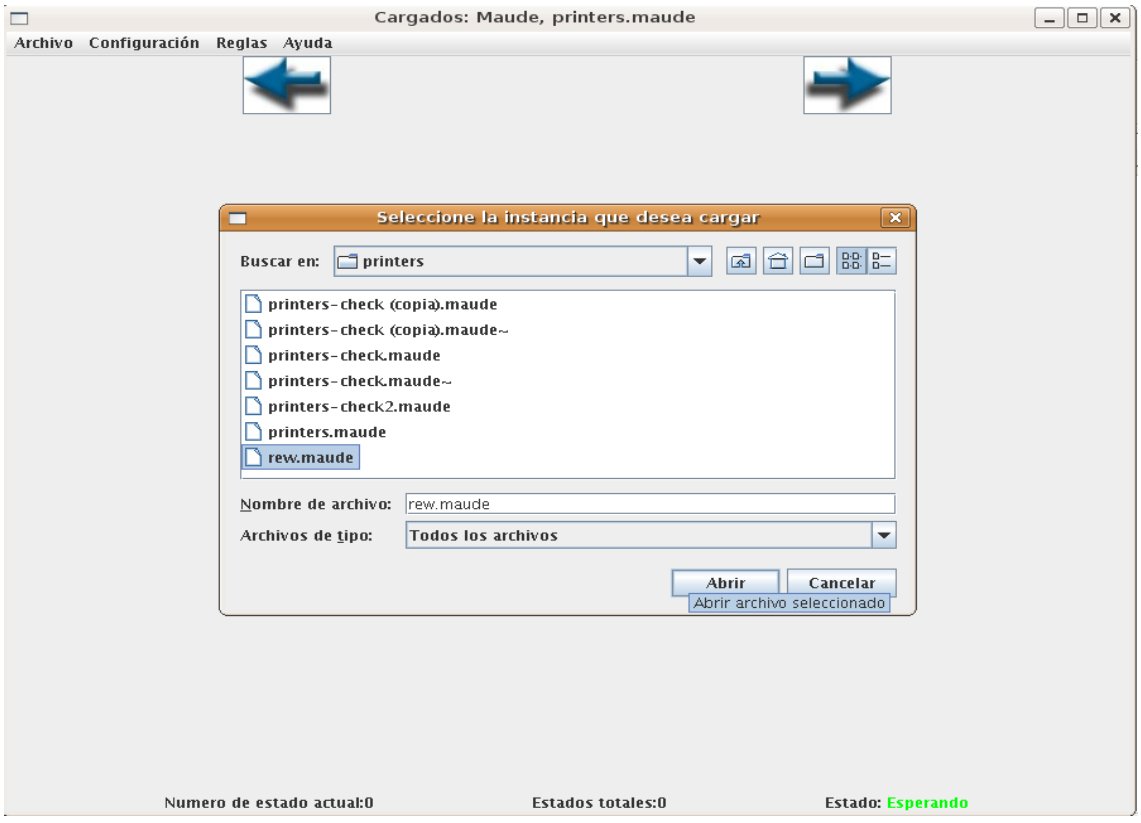


Como vemos arriba ya está cargado el Maude, ahora el siguiente paso es cargar el programa (en nuestro caso estamos probando con printers.maude) mediante el menú y eligiendo Cargar Programa mediante CTRL+a.



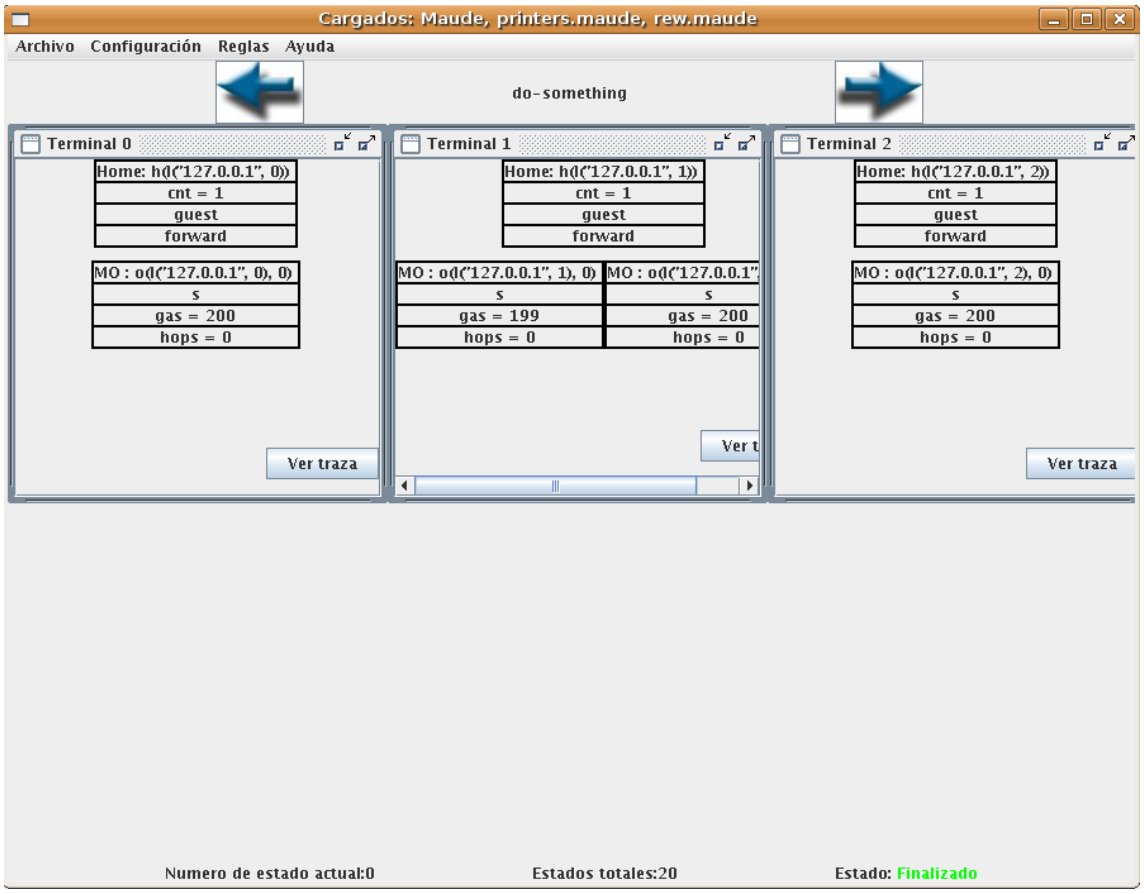


Una vez que hemos cargado el programa, queda cargar las instancias. En este caso cargamos el programa rew.maude. Debemos seleccionar del menú Cargar Instancia o pulsar CTRL+ i.

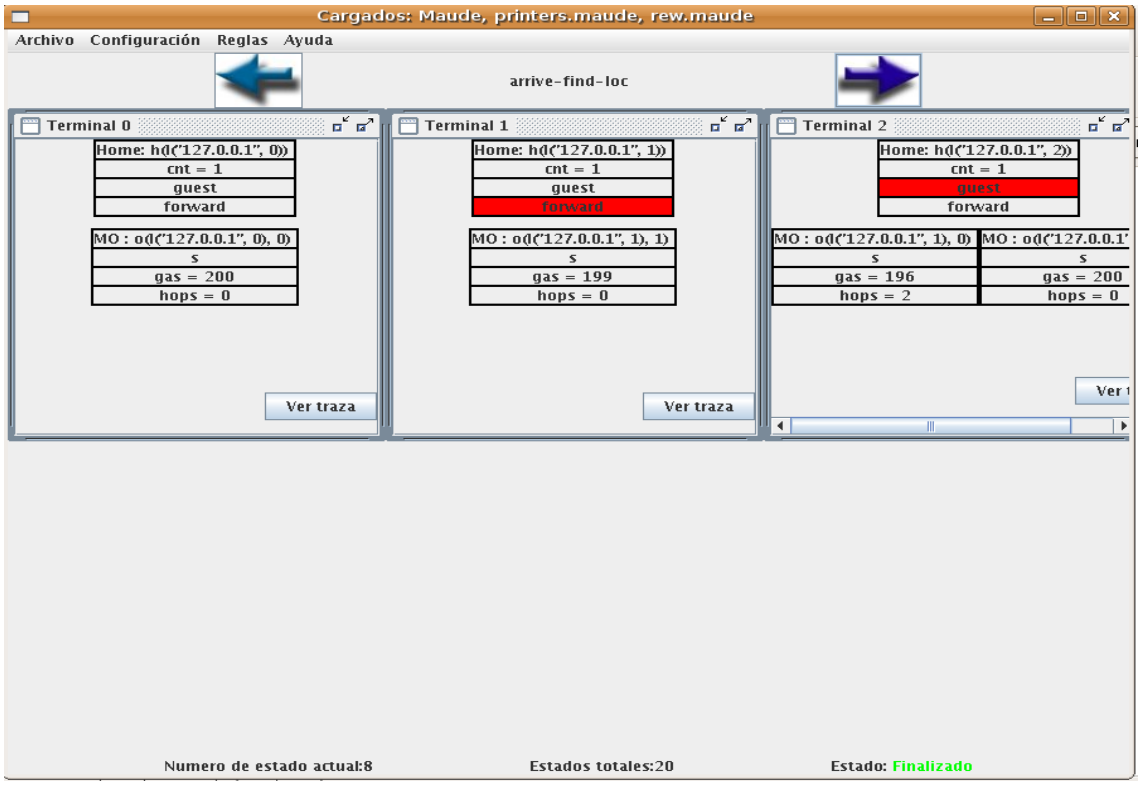


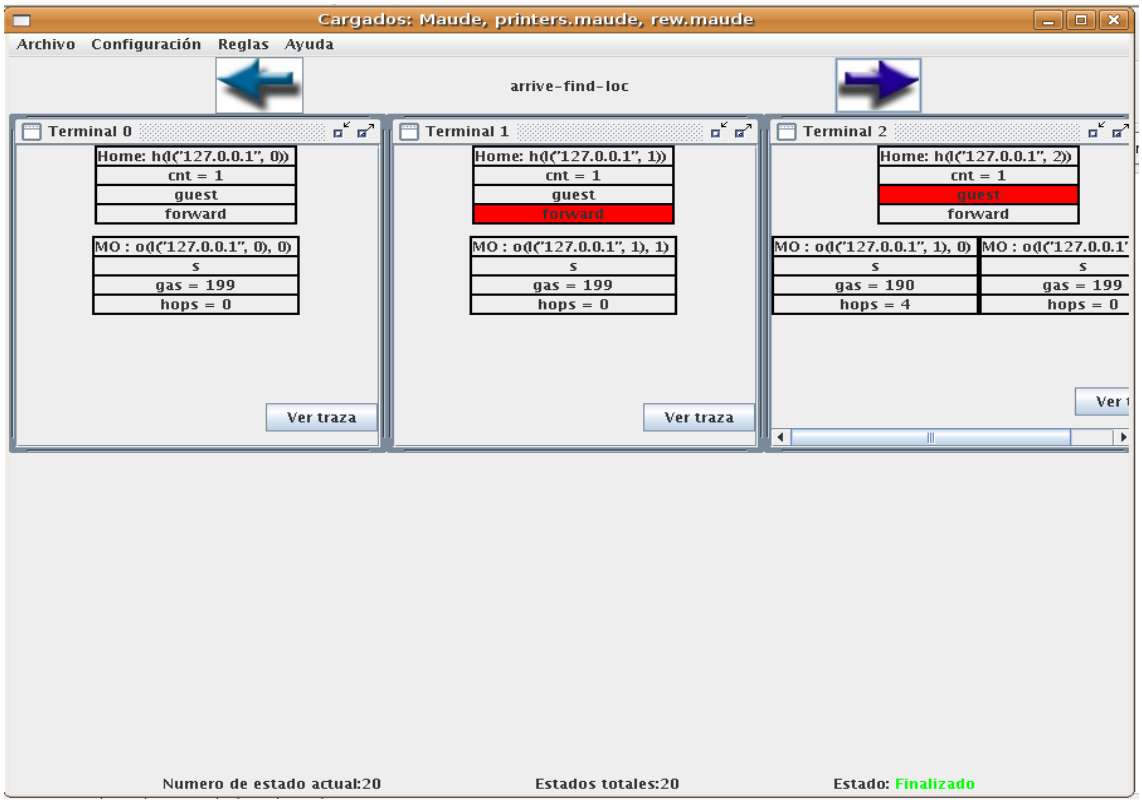
Una vez cargada el programa, la pantalla principal se divide en tantas subpantallas como procesos tenga el programa. Se generarán los estados, actualizándose el número de estados totales, el número de estado actual mostrará el estado en el que nos encontramos, y en estado se mostrará cargando si se están generando estados, y esperando si ha acabado de leer la traza disponible pero espera en caso de que llegue nueva traza. Con las flechas de izquierda y derecha podemos movernos por los distintos estados, desde el primero hasta el último, tanto pulsando la tecla con el ratón como manejándolo con teclado.

Como vemos en el título muestra las 3 cosas que hemos cargado (Maude, el programa printers y rew). Se muestran el estado 0 en este caso para cada terminal y la regla que se ha aplicado en ese estado:

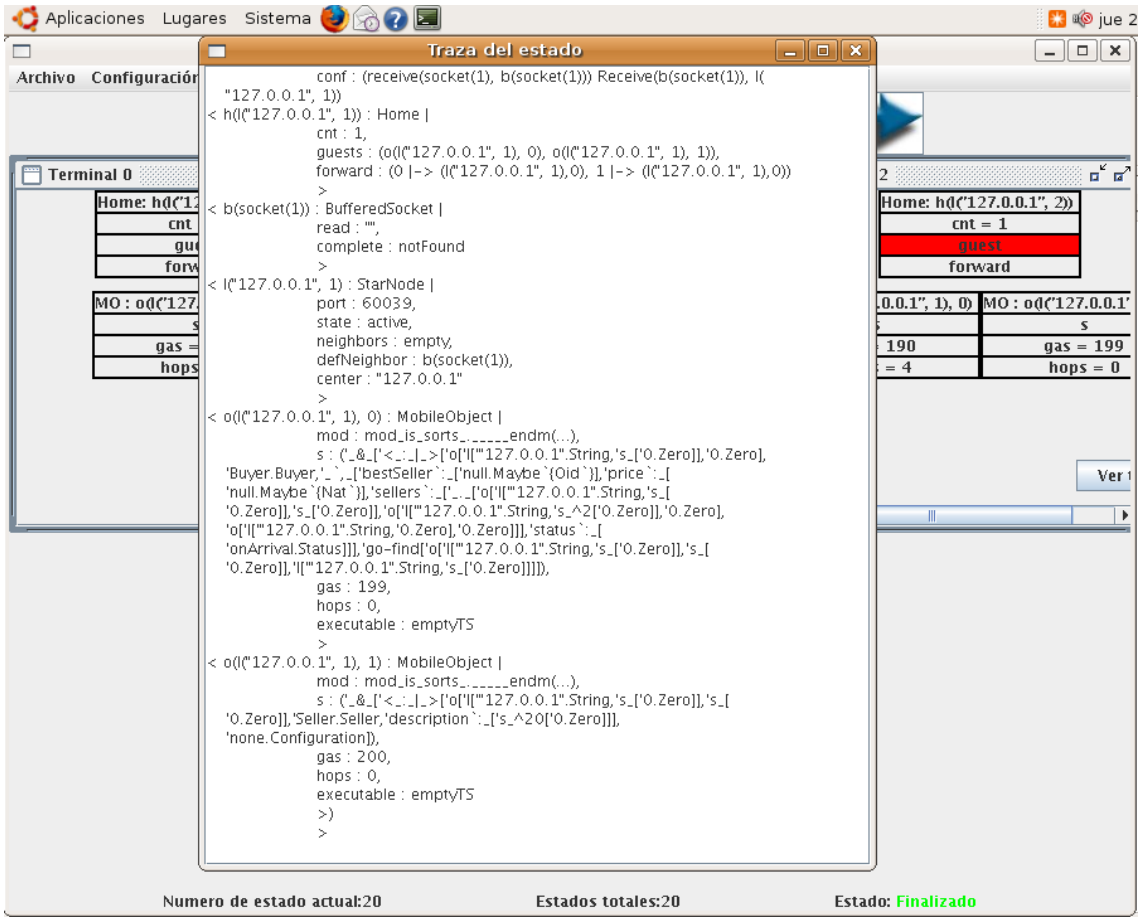


Si nos vamos moviendo por los estados vemos que se va actualizando el número de estado actual y también vemos resaltados en rojo aquellos atributos de cada terminal que en el anterior estado tenían otro valor distinto:





Como vemos en cada estado tenemos un botón ver traza que si lo pulsamos muestra la traza correspondiente al estado en el que nos encontramos:

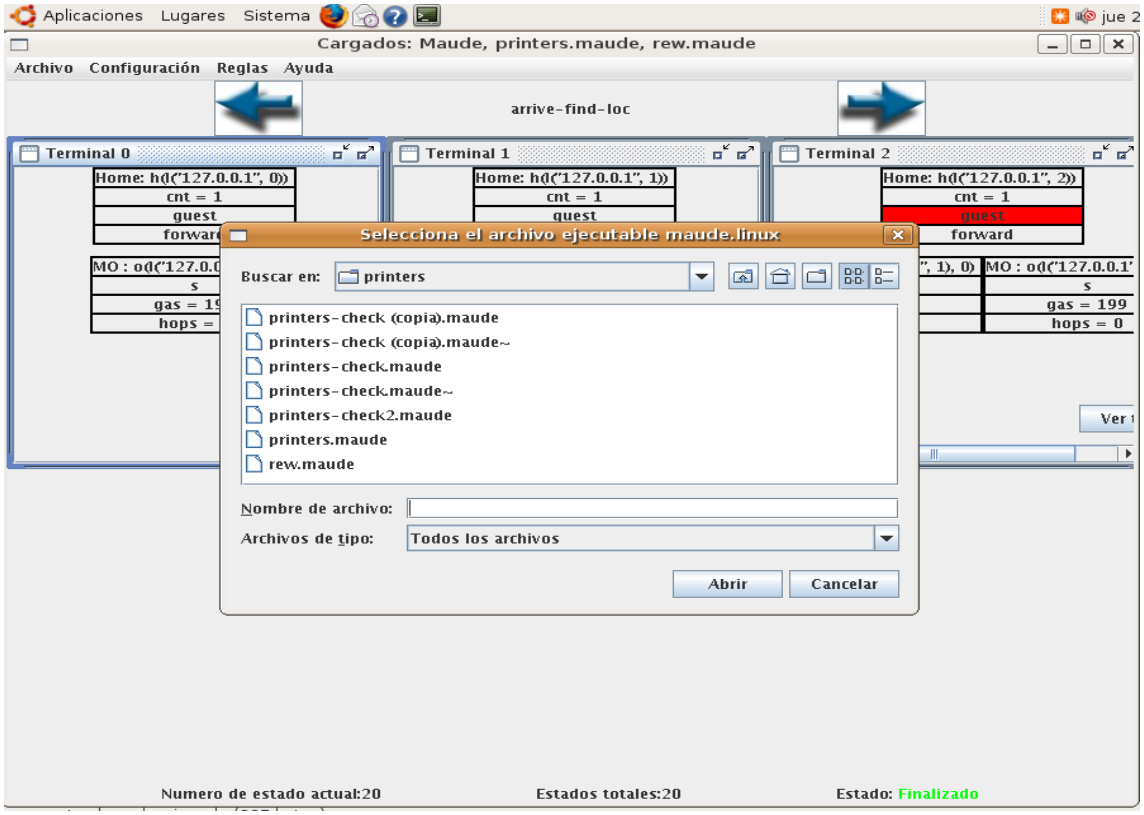


Algunas de las opciones que podemos hacer mediante la aplicación, es Resetear, es decir en una aplicación ya abierta, podemos volver a cargar otro programa y/o instancia diferente. Para ello en el menú archivo elegimos la opción Volver a Cargar Maude o pulsando CTRL +R.

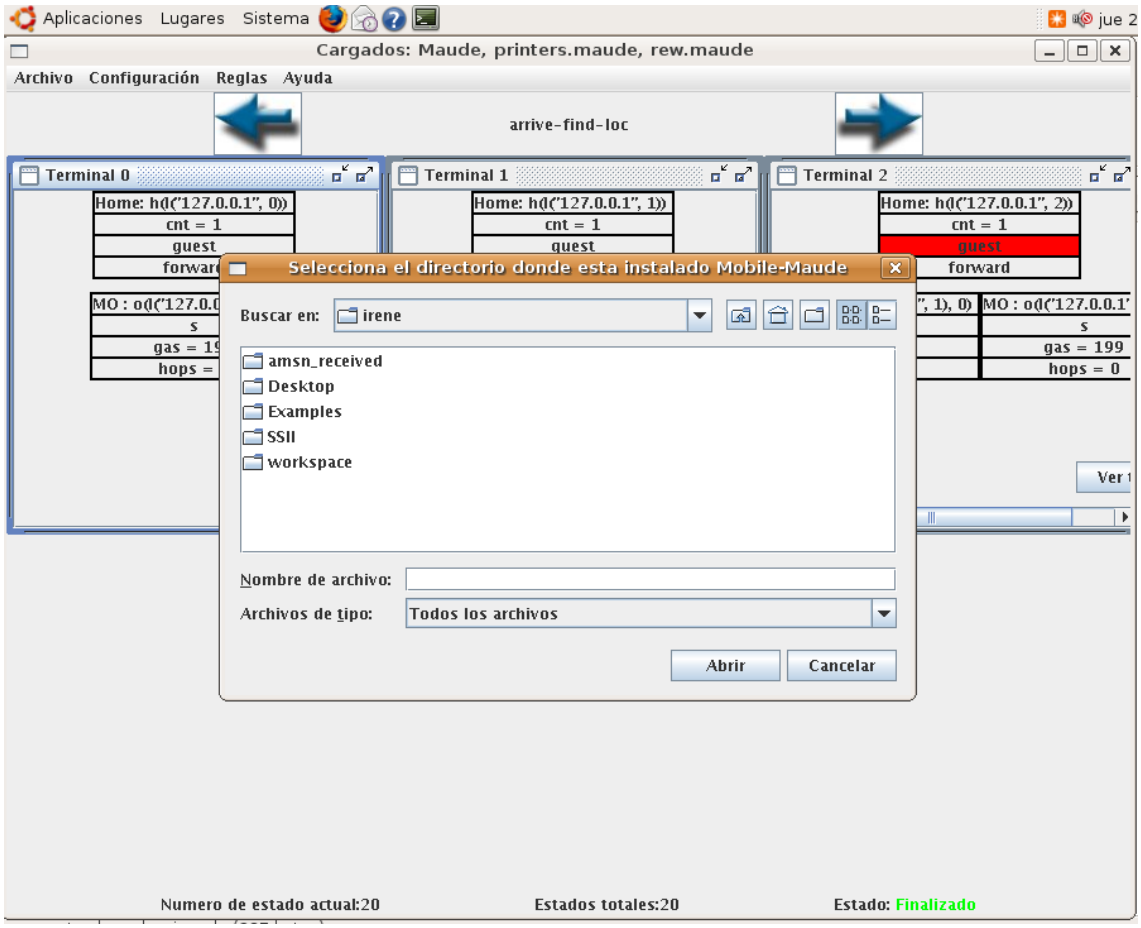
Otra de las opciones que tenemos es poder elegir las reglas que queremos mostrar o no. Para ello en el menú Reglas tenemos una lista de reglas fijas. En principio están todas seleccionadas pero si no queremos mostrar alguna simplemente tenemos que deseleccionarla y todos los estados referentes a esa regla no se mostrarán. Esto se puede cambiar tanto antes de cargar la instancia como una vez que ya este cargada.

Por último también permitimos cambiar la ruta donde se encuentra cargado el Maude y el Mobile Maude para la versión centralizada en tiempo de ejecución. En este caso no se modifica en el archivo config.properties sino que sólo valdría mientras esa aplicación este abierta.

Para ello si queremos cambiar la ruta del Maude seleccionamos la opción Configuración, luego Ruta Maude y ahí cambiar Ruta. Elegimos la nueva ruta que como hemos explicado en el párrafo anterior sólo es válida para esa aplicación.

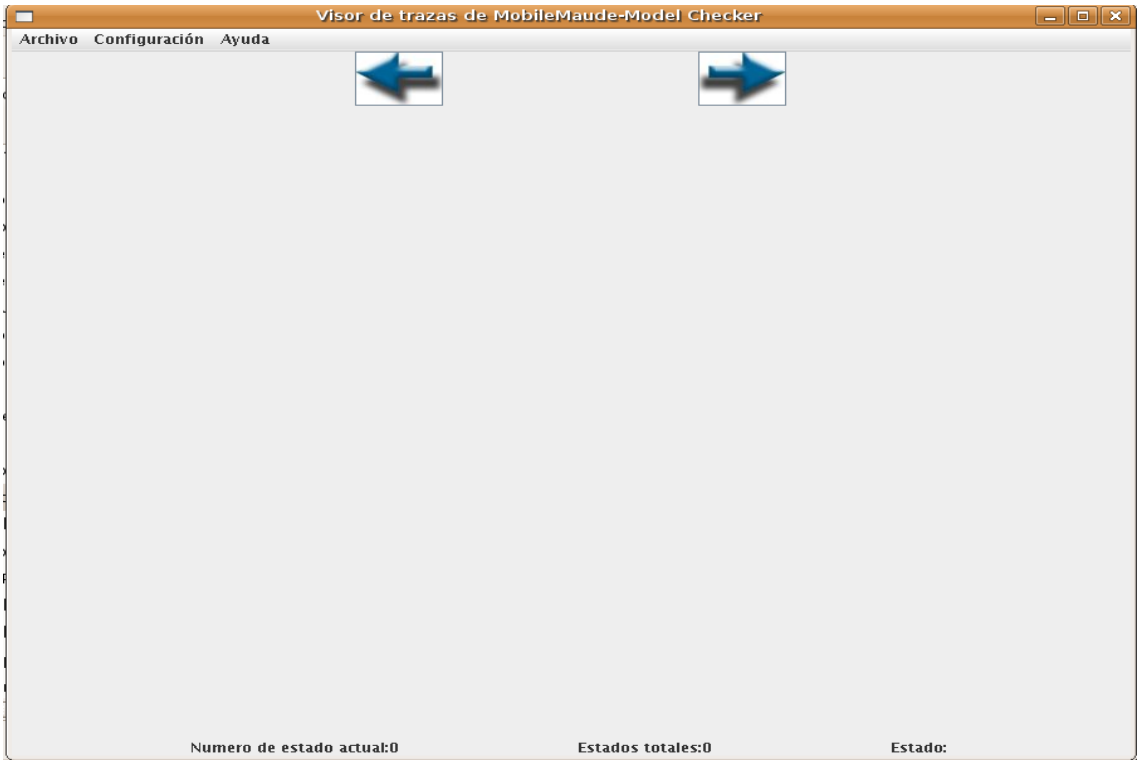


Si queremos cambiar la ruta de la versión centralizada de Mobile Maude, tenemos que seguir los mismo pasos pero seleccionando dentro del Menú Configuración la opción Mobile Maude.

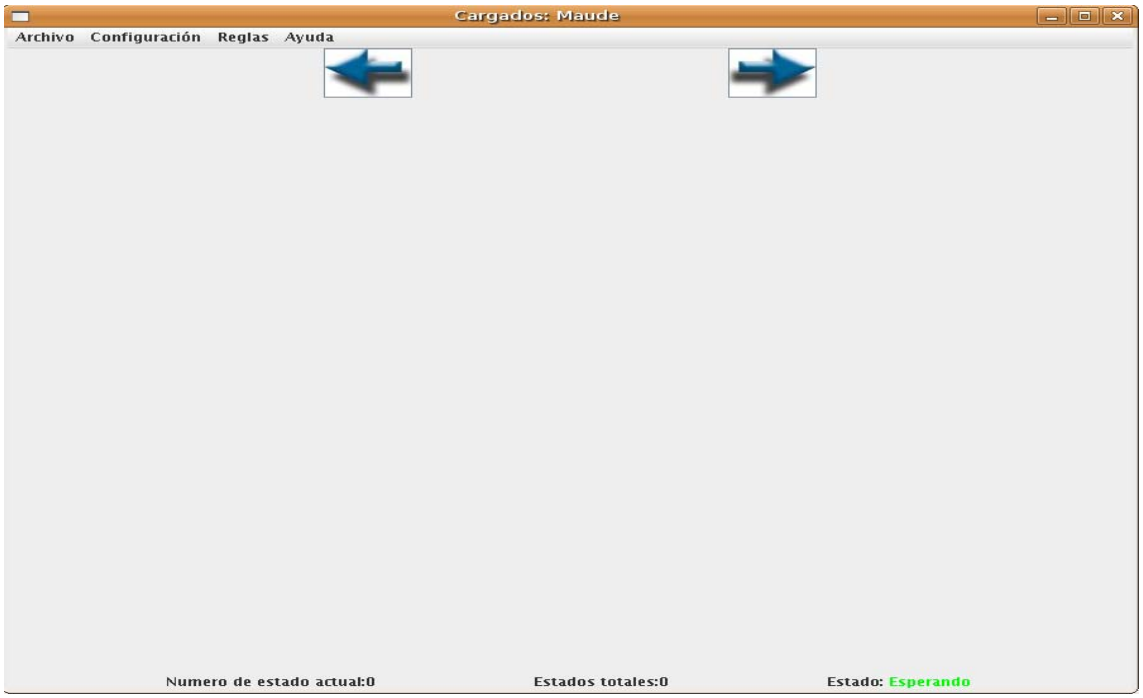


14.3. Funcionamiento del Model Checker

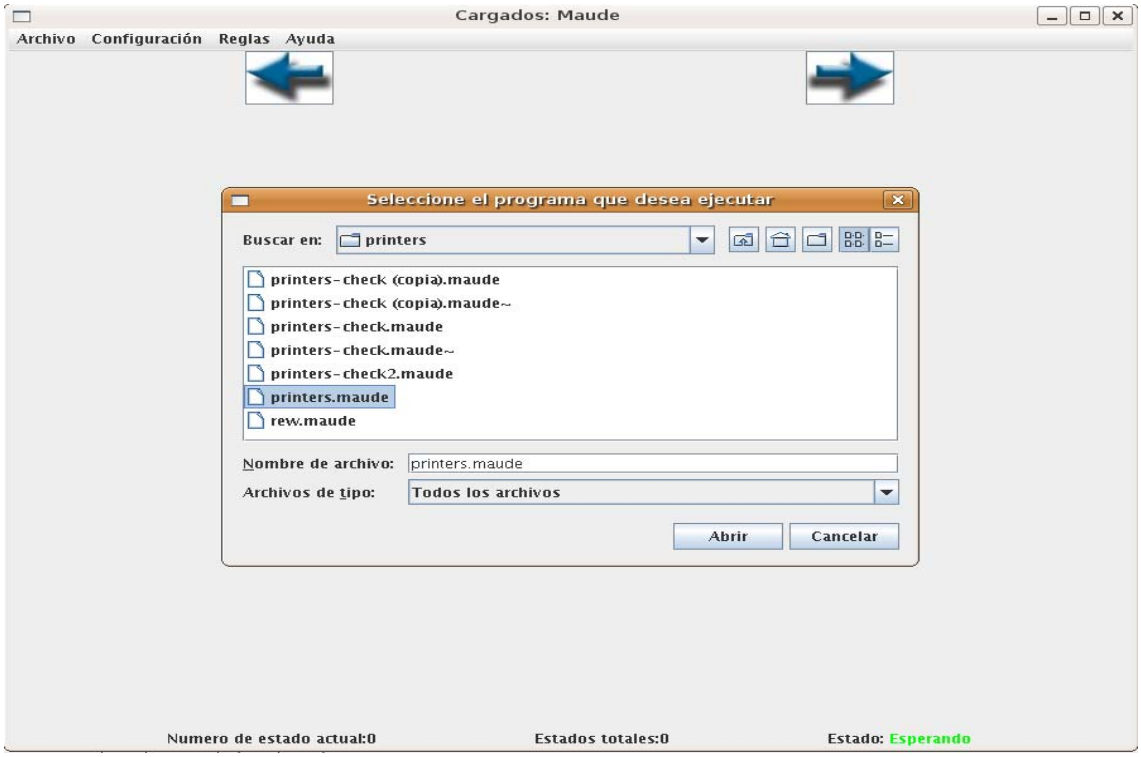
La pantalla principal del Model Checker es de la siguiente manera:

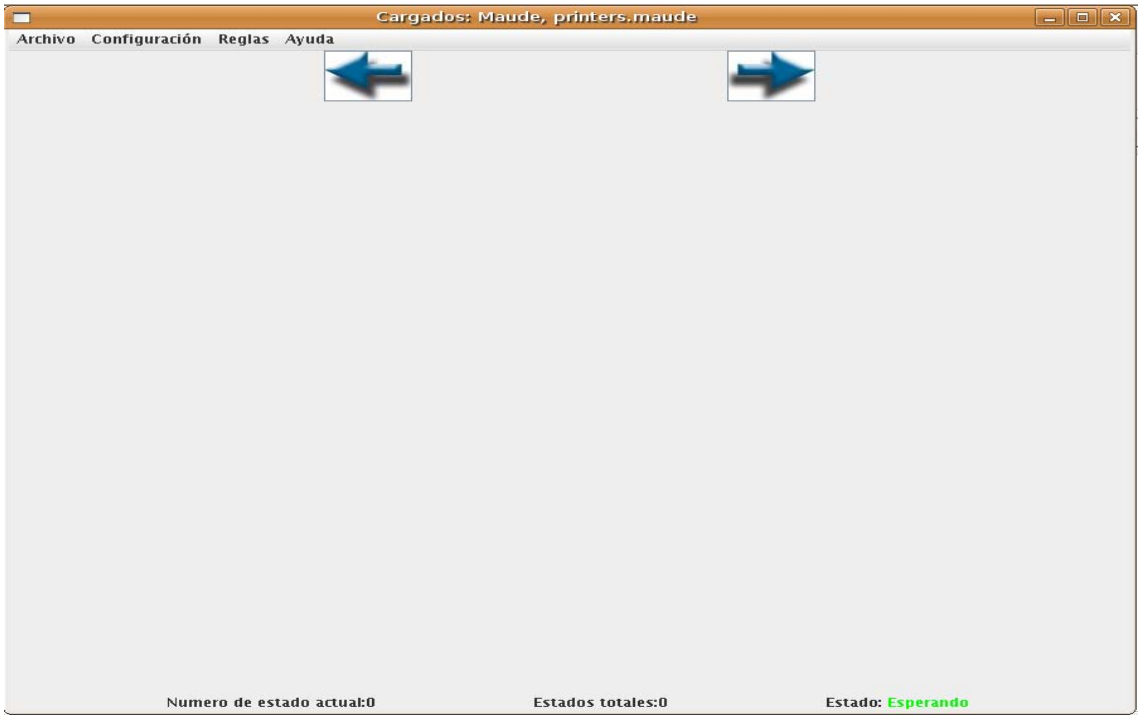


El siguiente paso es cargar Maude. Se puede hacer seleccionando en el menú Archivo la opción Cargar Mobile-Maude o mediante el teclado con CTRL+ m.

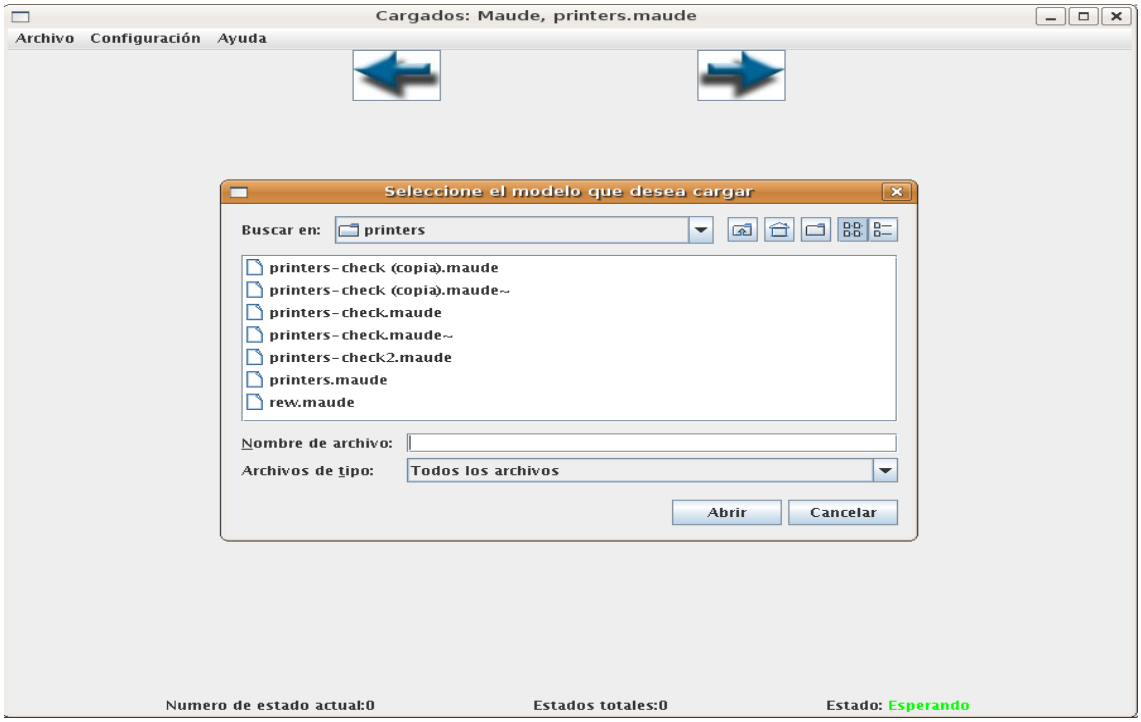


Como vemos arriba ya está cargado el Maude, ahora el siguiente paso es cargar el programa (en nuestro caso estamos probando con printers.maude) mediante el menú y eligiendo Cargar Programa o mediante CTRL+a.

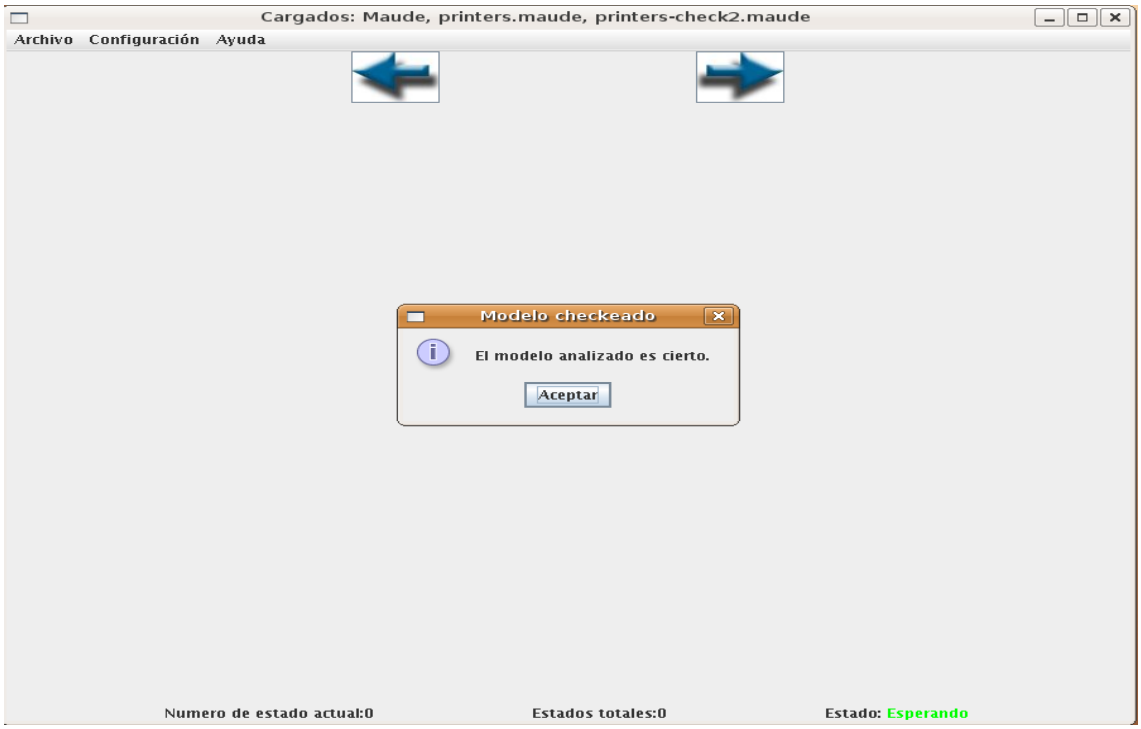


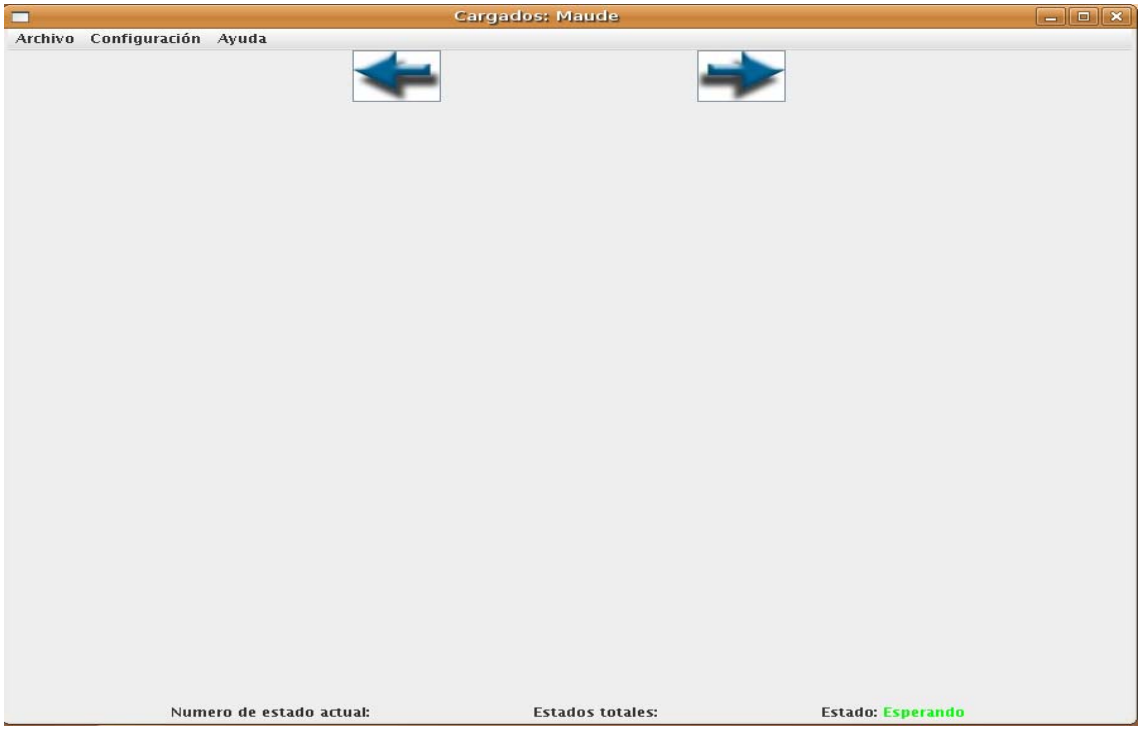


Una vez que hemos cargado el programa, queda cargar el programa que queremos chequear. En este caso cargamos el modelo printers-checker2.maude. Debemos seleccionar del menú Cargar Modelo o pulsar CTRL+ i. El modelo puede devolver cierto o falso:

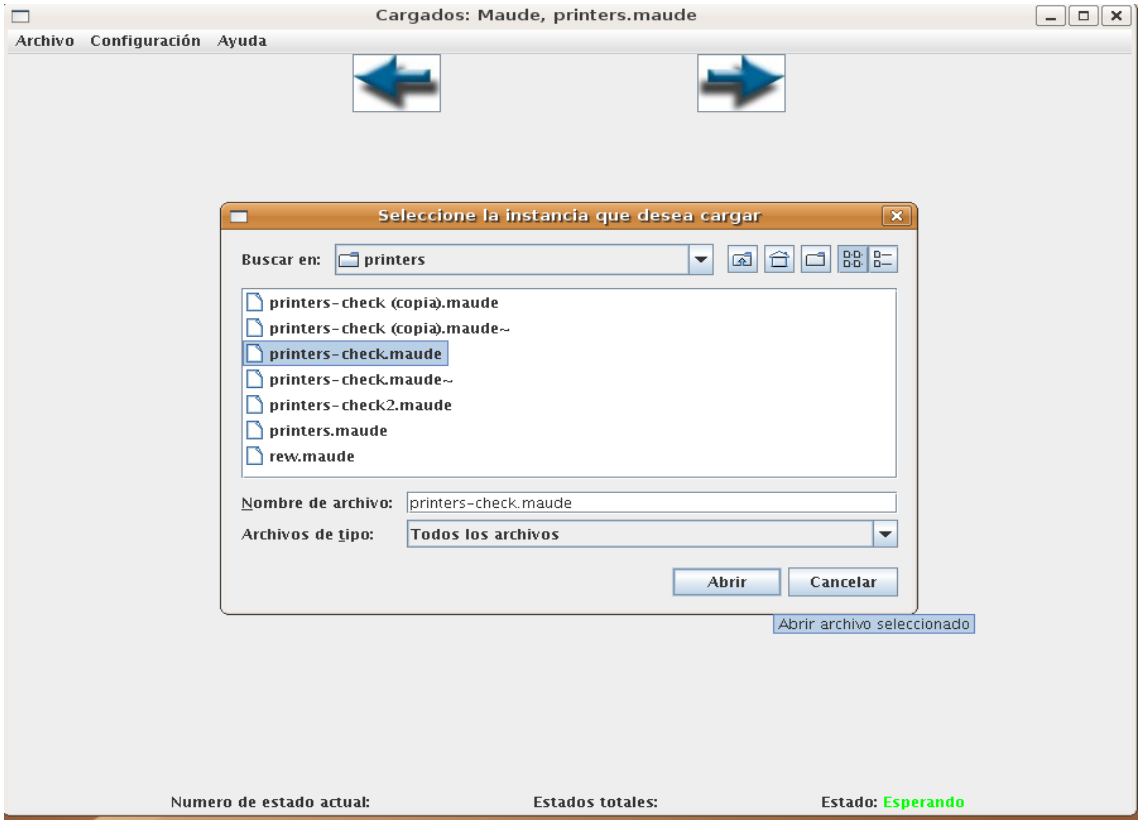


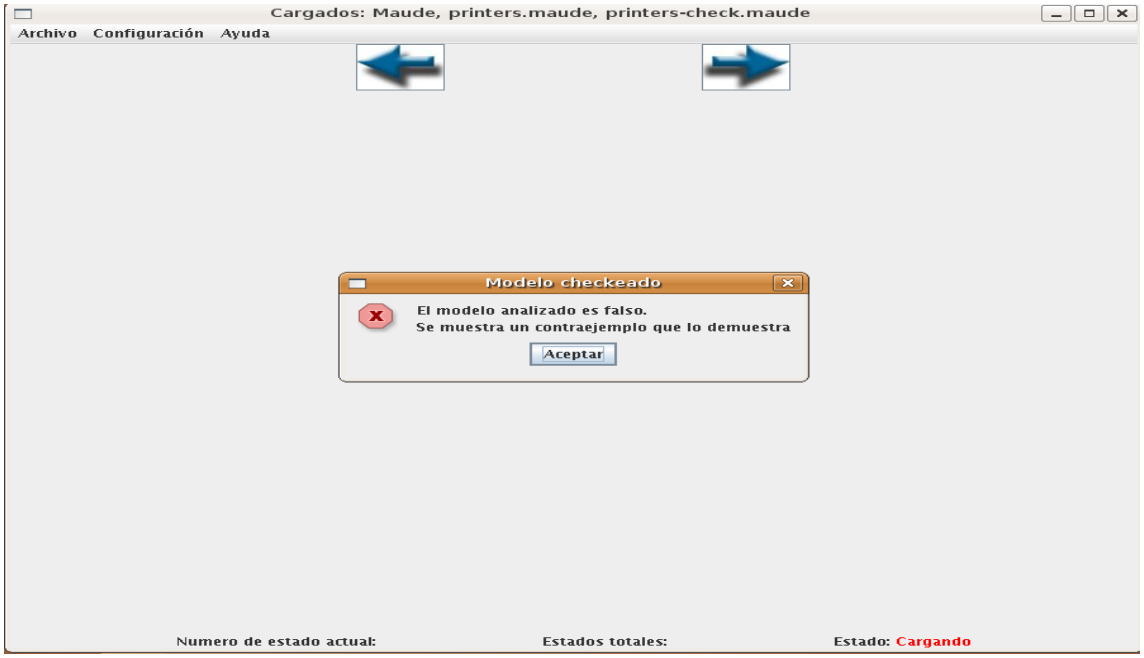
En caso de que el modelo sea cierto muestra un mensaje de que el modelo analizado es cierto.



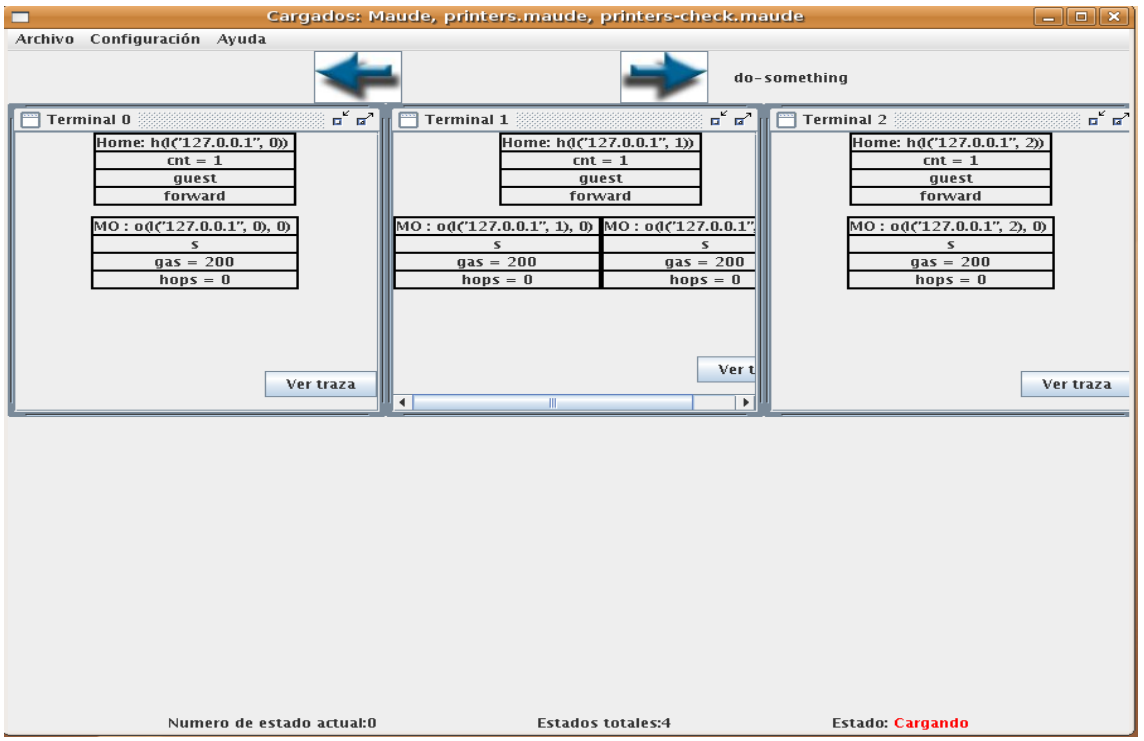


En caso de que sea falso muestra un mensaje de que es falso y devuelve un contraejemplo:



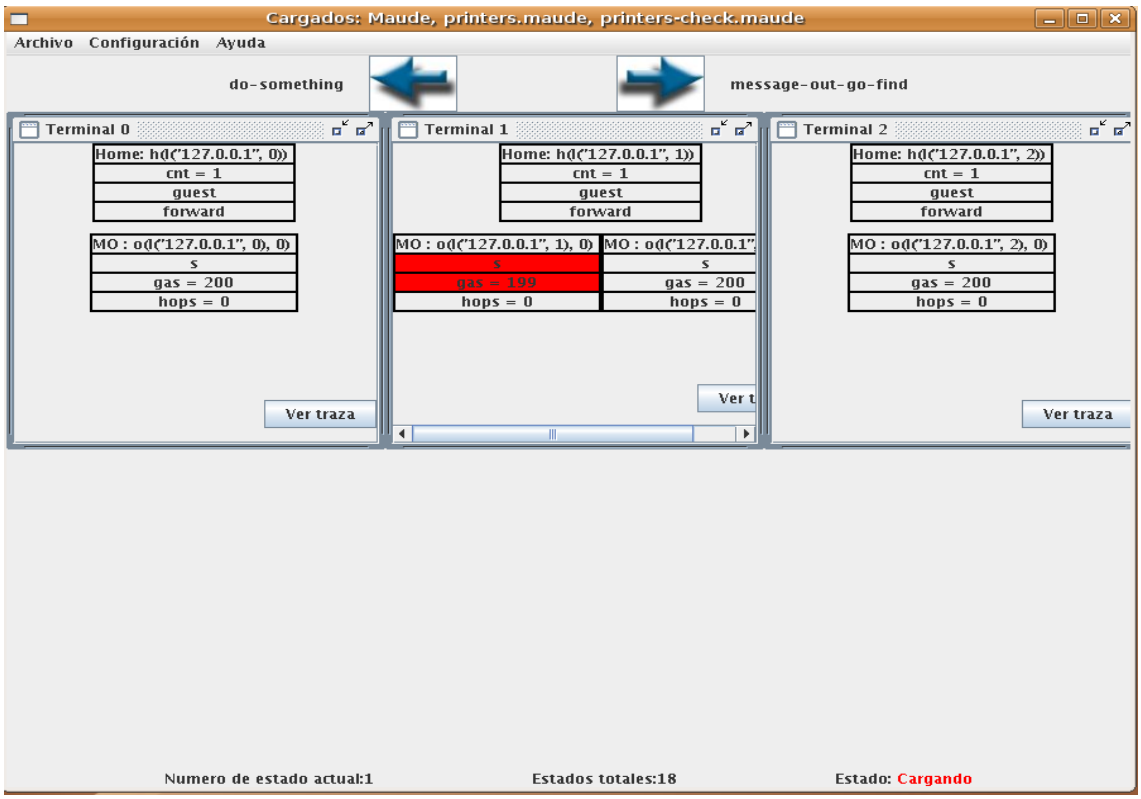


El contraejemplo tiene la misma estructura que la versión centralizada. Una vez cargada el programa, la pantalla principal se divide en tantas subpantallas como instancias tenga el programa. Se generarán los estados, actualizándose el número de estados totales, el número de estado actual mostrará el estado en el que nos encontramos, y en estado se mostrará cargando si se están generando estados.

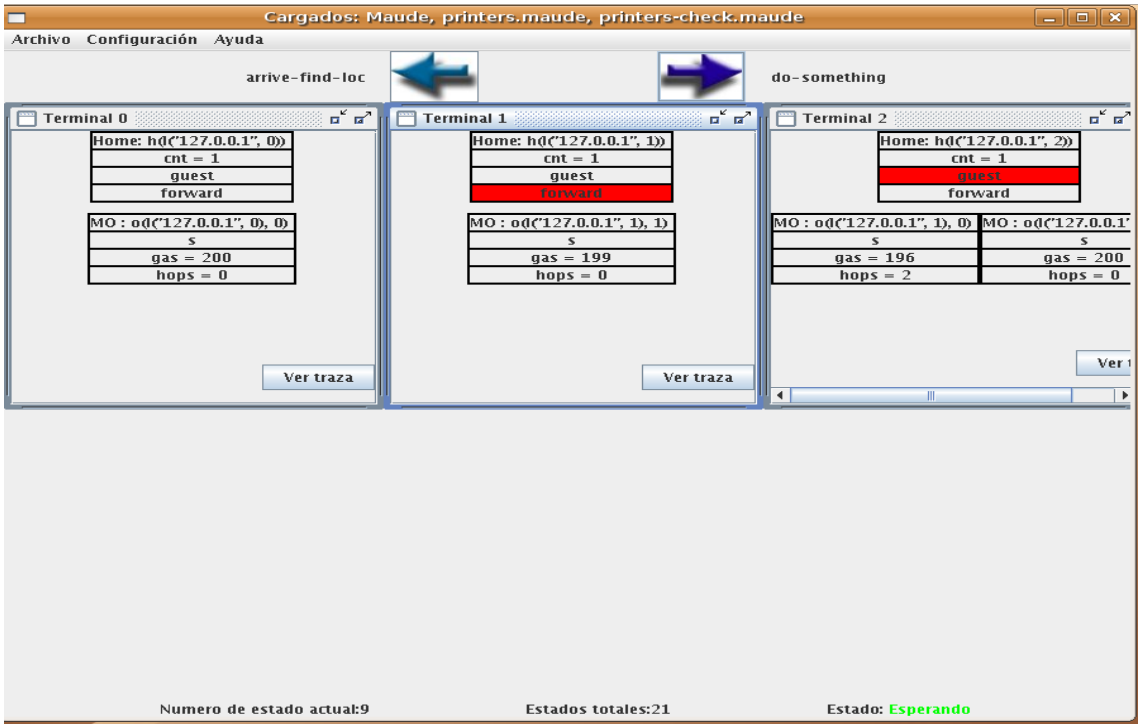


Con las flechas de izquierda y derecha podemos movernos por los distintos estados, desde el primero hasta el último, tanto pulsando la tecla con el ratón como manejándolo con teclado.

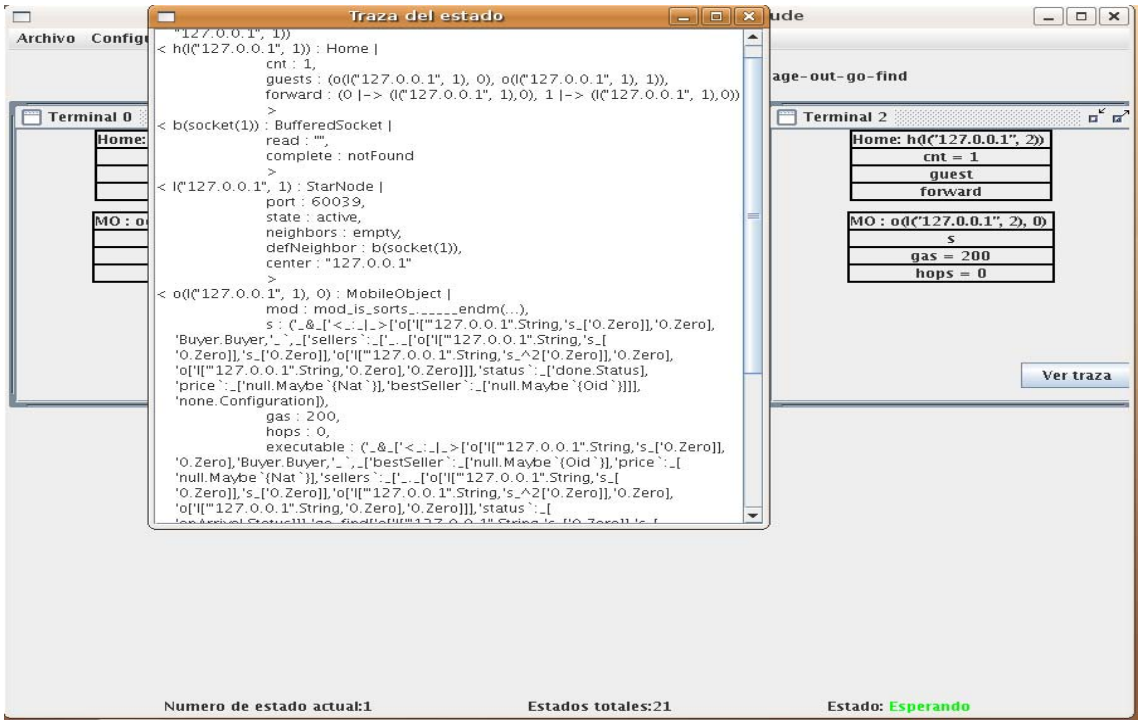
Como vemos en el título muestra las tres cosas que hemos cargado (Maude, el programa printers y printers-check). Se muestran el estado para cada terminal y la regla que se ha aplicado para llegar a ese estado y la regla que se aplicará para pasar al siguiente estado



Si nos vamos moviendo por los estados vemos que se va actualizando el número de estado actual y también vemos resaltados en rojo aquellos atributos de cada terminal que en el anterior estado tenían otro valor distinto:



Como vemos en cada estado tenemos un botón ver traza que si lo pulsamos muestra la traza correspondiente al estado en el que nos encontramos:

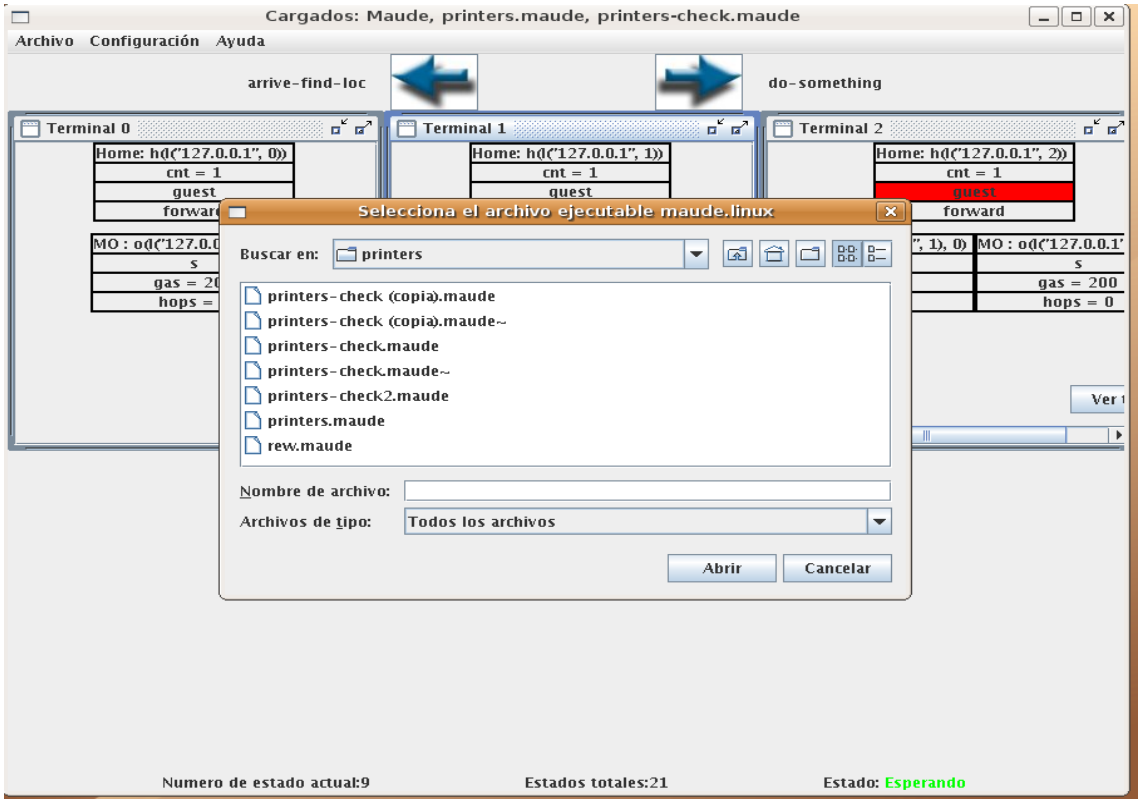


Algunas de las opciones que podemos hacer mediante la aplicación, es Resetear, es decir en una aplicación ya abierta, podemos volver a cargar otro programa y/o instancia diferente. Para ello en el menú archivo elegimos la opción Volver a Cargar Maude o pulsando CTRL +R.

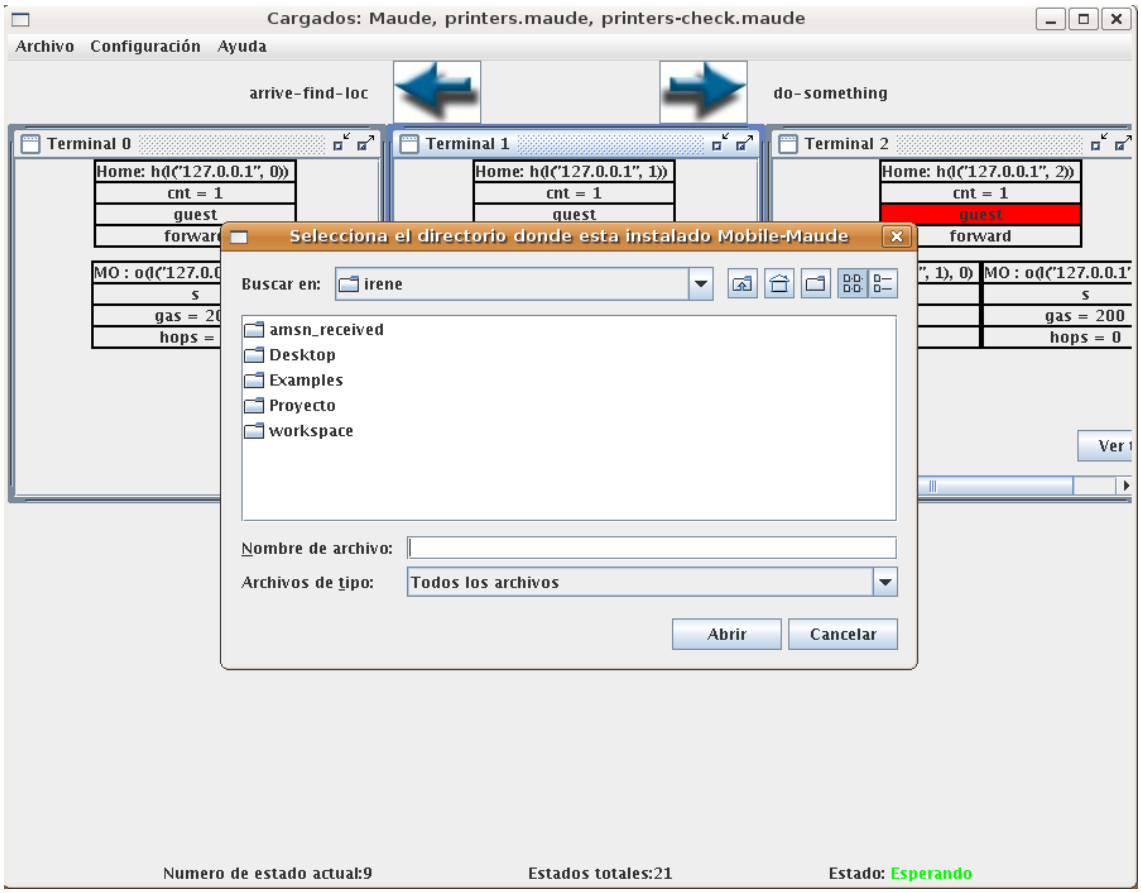
Otra de las opciones que tenemos es poder elegir las reglas que queremos mostrar o no. Para ello en el menú Reglas tenemos una lista de reglas fijas. En principio están todas seleccionadas pero si no queremos mostrar alguna simplemente tenemos que deseleccionarla y todos los estados referentes a esa regla no se mostrarán. Esto se puede cambiar tanto antes de cargar la instancia como una vez que ya este cargada.

Por último también permitimos cambiar la ruta donde se encuentra cargado el Maude y el Mobile Maude para la versión centralizada en tiempo de ejecución. En este caso no se modifica en el archivo config.properties sino que sólo valdría mientras esa aplicación este abierta.

Para ello si queremos cambiar la ruta del Maude seleccionamos la opción Configuración, luego Ruta Maude y ahí cambiar Ruta. Elegimos la nueva ruta que como hemos explicado en el párrafo anterior sólo es válida para esa aplicación.



Si queremos cambiar la ruta de la versión distribuida de Mobile Maude, tenemos que seguir los mismo pasos pero seleccionando dentro del Menú Configuración la opción Mobile Maude.



15. CONCLUSIÓN

La realización de este proyecto nos ha servido para conocer un lenguaje desconocido para nosotros, Maude, y su versión para objetos móviles, que es Mobile Maude. Al principio nos ha costado familiarizarnos un poco con el lenguaje y la traza resultante de su ejecución, pero a medida que hemos ido leyendo la documentación que se nos facilitó hemos podido entender la traza y así sacar la información que necesitábamos mostrar de forma gráfica.

Otro de los aspectos importantes es que para que Maude funcionara correctamente teníamos que desarrollarlo en Linux o MAC (en nuestro caso elegimos Linux), sistemas distintos al que diariamente usamos como usuarios de PC por lo que fue una dificultad añadida, pero de la que hemos sacado una experiencia positiva, ya que nos ha valido para ganar conocimientos en un sistema operativo que se usa actualmente bastante y no cerrarnos así puertas de cara a la vida laboral o incluso al propio uso como usuarios.

También hemos ampliado nuestros conocimientos del lenguaje JAVA, ya que, aunque teníamos experiencia en desarrollo de prácticas, todas eran guiadas, por lo que los conocimientos adquiridos eran muy delimitados. En muchos momentos hemos tenido que investigar por nuestra cuenta y eso nos ha servido tanto para aumentar esos conocimientos como para poder coger soltura sin la necesidad de tener un profesor que nos guiará totalmente en su realización, aunque también es verdad que en algunos momentos y sobre todo, referente a dudas sobre Maude, hemos acudido a nuestro Tutor, el cuál siempre nos ha ayudado sin problemas y que gracias a él el desarrollo de este proyecto ha sido posible.

Otra ventaja que hemos sacado ha sido que hemos aprendido bastantes nuevos conceptos que a lo largo de la carrera no se aprenden, no sólo sobre nuevas tecnologías o de aumentar conocimientos sobre cosas que ya sabíamos, sino conceptos que van más allá de lo teórico, tales como el saber desenvolverse en situaciones difíciles, tener que buscar información por nuestra cuenta, la coordinación entre los miembros del equipo, el entendimiento con el Tutor y cumplir los objetivos que nos ha marcado en una fecha concreta pero a largo plazo, en la que tienes que planificar el desarrollo entre el grupo.

En resumen todas estas cosas que no te enseñan en un asignatura de la carrera son las que en el fondo nos van a servir en nuestro trabajo, ya que nadie va a estar guiándote todo el rato, sino que te va a poner un fecha y un objetivo que tienes que cumplir y creemos que gracias a este proyecto y a nuestro Tutor hemos valorado y aprendido y con las que hemos logrado su realización de este proyecto.

16. LISTA DE PALABRAS CLAVE

- Maude
- Mobile Maude
- Model Checker
- Traza de Maude
- Interfaz Gráfica
- Objetos Móviles
- Java
- Linux y MAC

17. BIBLIOGRAFIA

- Durán F., Riesco A. y Verdejo A.: **A distributed implementation of Mobile Maude**. WRLA 2006.

- Palomino M., Martí-Oliet N. y Verdejo A.: **Playing with Maude**. Elsevier Science, 2005.

- **Complete List of Maude Commands** (Capítulo 22). **All about Maude: A high-performance logical framework.** Springer, 2007.

- **Debugging and Troubleshooting**. (Capítulo 23) **All about Maude: A high-performance logical framework.** Springer, 2007.

- Riesco A.: **Distributed and mobile applications in Maude.** Universidad Complutense de Madrid, 2007.

- Martí-Oliet. Proyecto investigación científica**. "Metalenguajes para el Diseño y Análisis Integrado de Sistemas Móviles y Distribuidos**". Universidad Complutense de Madrid, 2002

- **The Java Tutorial.** Sun Microsystems. http://java.sun.com

- Deitel, H., y Deitel, P: **Java, how to program**. Prentice Hall, 1998.

- Sánchez, J., Huecas, G., Fernández, B. y Moreno, P.: **Java 2**. McGraw-Hill 2001.

- Alfredo Muñoz Andrades. Proyecto fin de carrera**. "Maude Workstation: Entorno de Programación para el Lenguaje de Especificación Algebraica Maude**". Universidad de Málaga.

Autorizamos a la Universidad Complutense a difundir y utilizar con fines académicos, no comerciales y mencionando expresamente a sus autores, tanto la propia memoria, como el código, la documentación y/o el prototipo desarrollado.

Firmado

Irene González Palomar

Diego González Rodríguez

Luis Julián Plaza Fernández