Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Desarrollo de un depurador reversible para programas Prolog

Trabajo Fin de Grado

Grado en Ingeniería Informática

Autor: Vicente Fructuoso Chofré

Tutor: Germán Francisco Vidal Oriola

Curso 2020-2021

Resumen

Este TFG presenta la realización de un depurador reversible de Prolog que permitirá a los usuarios la carga y visualización de los distintos estados por los que pasa el programa para la obtención de una solución tales como las llamadas, fallos, llamadas de *backtracking* y obtención de soluciones alternativas. Para esto se utilizará una aplicación destinada principalmente al estudiante de informática que buscará un uso fácil y cómodo con el usuario que le ayude con el uso de la aplicación, pudiendo moverse cómodamente entre las distintas llamadas proporcionadas. Utilizando el mismo Prolog para las distintas llamadas y Java para la creación de la interfaz mostraremos también la integración entre estos dos lenguajes.

Palabras clave: Prolog, Java, llamada, salida, paso

Abstract

This TFG presents the realization of a reversible Prolog debugger that will allow users to load and visualize the different states through which the program passes to obtain a solution such as calls, failures, backtracking calls and obtaining solutions alternatives. For this, an application will be used mainly intended for the computer science student who will look for an easy and comfortable use with the user that helps him with the use of the application, being able to move comfortably between the different calls provided. Using the same Prolog for the different calls and Java for creating the interface, we will also show the integration between these two languages.

Keywords: Prolog, Java, call, exit, step.

Dedicatoria y agradecimientos

A mi familia por su apoyo en la redacción de este trabajo.

A mi tutor por estar siempre cuando se le necesitaba y su ayuda constante.

A los profesores, médicos y demás personal que han trabajado para que este año tan tristemente atípico fuera lo más normal posible.

Tabla de contenidos

1. Introducción

La creciente evolución de la sociedad humana hacia un futuro informático ha hecho que la profesión del programador informático se convierta en una oportunidad de futuro indiscutible y que la figura del programador sea necesitada tanto por empresas de mayor o menor envergadura como por gobiernos de todo tipo.

Ya sea en la creación de aplicaciones para el uso cotidiano o para uso empresarial o la implantación de una defensa informática en los aparatos gubernamentales hasta la simple confección de un programa de un estudiante hecho para aprender, el programador siempre hará uso de una herramienta indispensable, un lenguaje de programación.

El mayor porcentaje de uso de un lenguaje frente a otro ya sea en un entorno laboral o educativo se basa en la potencia y beneficios que un lenguaje puede aportar en su utilización como también la facilidad de uso y aprendizaje que éste presenta. Por esto algunos lenguajes como Java se han convertido en los más utilizados en casi todos los ámbitos mientras que otros han quedados más abandonados. La dificultad de estos últimos lenguajes ha hecho que sean menos utilizados en centros educativos por lo que los programadores expertos en ellos son más escasos y deseados cuando son necesitados.

1.1 Motivación general

En este documento mostraremos la realización de un depurador de programas de Prolog que nos permitirá ver la ejecución paso a paso de una llamada. A partir de un código Prolog aportado por el tutor, se pretende desarrollar una interfaz gráfica amigable.

La principal motivación de este proyecto se basa por tanto en la creación de una herramienta sencilla que ayude a alumnos o investigadores interesados en este lenguaje para que puedan conocer mejor la forma en que se ejecutan los programas en este lenguaje de programación.

1.2 Motivación personal

Las principales razones para hacer este TFG se encuentran en mi interés por hacer uso de las habilidades de programación que he ido adquiriendo a lo largo de la carrera. Además, también valoré la posibilidad de aprender un lenguaje de programación que no controlaba.

1.3 Objetivos generales y específicos

En este trabajo buscaremos los siguientes objetivos:

- Desarrollar una interfaz en Java para una aplicación Prolog.
- La aplicación debe mostrar los pasos de ejecución en la obtención de la solución a una llamada.
- Permitir al usuario recuperar el paso anterior de la ejecución.
- Crear una aplicación sencilla y ágil de manejar para los estudiantes del lenguaje Prolog.

1.4 Impacto

El principal impacto que buscamos es en el aprendizaje de Prolog ya que el depurador ayudará a los alumnos en su adquisición de conocimiento sobre este lenguaje y les motivará a indagar más sobre las cualidades que éste presenta y utilizarlo en futuros proyectos. El depurador puede introducirse a los propios alumnos o a través de los profesores mediante alguna asignatura de programación. Si bien estos son el principal público objetivo, nuestra aplicación puede usarse por cualquier persona interesada en el lenguaje Prolog y en aumentar sus conocimientos en la materia.

1.5 Estructura

En primer lugar hablaremos del estado de la tecnología actual, comentando la opción elegida, explicando sus ventajas y la razón por la cual la hemos elegido frente a las demás alternativas.

Después pasaremos a hablar del análisis del problema, donde explicaremos las concepciones iniciales del trabajo, como su planificación enunciando los requisitos necesarios para su implementación.

En la parte del diseño de la solución explicaremos los casos de uso y el desarrollo de las partes que componen la solución. Mientras que en la parte de desarrollo explicaremos la implementación concreta de estas partes.

Acabaremos con los resultados de las pruebas y una conclusión sobre los resultados obtenidos.

2. Estado del arte

En este apartado presentaremos el estado actual de las herramientas similares a la presentada en este trabajo además de diversas herramientas para la implementación de la aplicación tanto la elegida como alternativas.

2.1 Prolog

Para comenzar este trabajo, será necesario una pequeña introducción al lenguaje Prolog. Prolog (Colmerauer,Roussel;1970) es un lenguaje de programación que se basa en la lógica de primer orden o lógica de predicados, en el que los programas están formados por un conjunto de hechos y reglas. El comportamiento básico de Prolog es el siguiente: Ante una determinada consulta al programa, se utilizarán estas reglas y hechos para intentar demostrar la veracidad o falsedad de la consulta.

Un programa Prolog está formado por predicados que se definen por dos características: Su nombre, con el cual se identifica, y su aridad o número de argumentos. Por ejemplo, español/1 y español/3 son predicados distintos debido al número de argumentos que aceptan aunque se llamen igual. Los predicados pueden estar formados por una o más clausulas; por ejemplo, podemos tener una cláusula `español(catalán)` pero también una cláusula `español(madrileño)`. Estas dos clausulas serían lo que antes hemos definido como hechos: afirmaciones que consideramos ciertas. Las reglas son construcciones lógicas formadas por una cabeza y un cuerpo como, por ejemplo:

```
español_costero(X) :- costa(X),español(X).
```

Esta regla se puede interpretar como sigue: X es un español costero si y solo si X es español y de un origen con costa. Como podemos observar, la parte izquierda de la regla será cierta cuando la parte derecha también lo sea.

El mecanismo de búsqueda de soluciones es el siguiente: tomemos los hechos y la regla anterior y añadámosle dos hechos más: `costa(napolitano)` y `costa(catalán)`. Si preguntamos por un hecho, un intérprete nos responderá directamente como cierto o falso dependiendo de la existencia de ese hecho. Si en cambio preguntáramos que valor X puede cumplir el ser un español costero el programa buscaría primero qué hechos cumplen el tener costa. En este caso lo cumplirían napolitano y catalán. En este momento, nos encontramos ante un punto de elección. Sin embargo, al elegir la opción X = napolitano la segunda llamada, `español(napolitano)` no tendría éxito. En este momento se efectúa un mecanismo de vuelta atrás o *backtracking* en el cual deshacemos todo el trabajo ejecutado hasta el último punto de elección y tomamos otra alternativa. Si ahora elegimos X = catalán, `español(catalán)` si tendría éxito. Por lo tanto podemos devolver catalán como una solución.

Pese a que Prolog puede ser relevante en el ámbito de la inteligencia artificial, en términos generales su uso es minoritario hoy en día, a menudo restringido al ámbito académico.

2.2 Depurador SWI-Prolog

Actualmente en el mercado no se encuentran muchos depuradores que utilicen Prolog como base. El principal que encontramos es el que proporciona SWI-Prolog (Wielemaker; 1987) como depurador. En la figura 2.1 podemos ver una imagen del intérprete de SWI-Prolog.

***Figura 2.1**: Imagen intérprete.*

Para activar el depurador, usaremos el predicado trace. Esto nos permitirá observar paso a paso la ejecución de una llamada. En la figura 2.2 podemos ver un ejemplo.

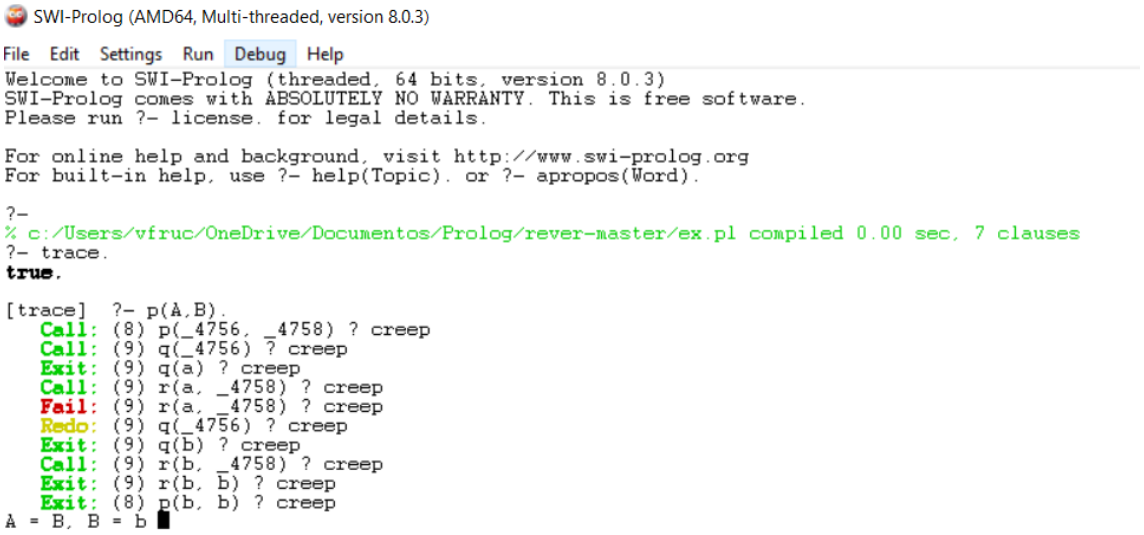


***Figura 2.2**: Imagen ejecución.*

Como podemos ver el depurador devuelve la ejecución del programa. Para llevarla a cabo, Prolog utiliza un modelo conocido como modelo de Byrd que se basa en cuatro tipos de eventos:

- Call: Sucede cuando se intenta satisfacer un objetivo.
- Exit: Sucede cuando algún objetivo se cumple.
- Fail: Sucede cuando algún objetivo falla.
- Redo: Suceso de *backtraking*, Prolog revierte la ejecución del programa hasta este punto para intentar cumplir el objetivo con parámetros diferentes.

Utilizando estos eventos el depurador realizará la ejecución paso a paso de los programas Prolog. La versión rever del depurador así como nuestra interfaz utilizaran también este modelo.

El depurador de SWI-Prolog permite además realizar diversas acciones sobre la ejecución. La figura 2.3 enseña unas tablas con algunos de los comandos.

Nombre	Comando	Descripción
Abort	a	Aborta la ejecución de Prolog
Break	b	Abre un nuevo hilo de Prolog
Creep	c	Lleva a cabo un paso de ejecución
Exit	e	Cierra SWI-Prolog
Fail	f	Fuerza el fallo del siguiente objetivo
Find	/	Continua la ejecución hasta que determinado patrón es encontrado
Ignore	i	Ignora el siguiente objetivo pretendiendo que a dado acierto

Leap	l	Continua la ejecución hasta el siguiente *spy point*
No debug	n	Continuar ejecución en modo *no debug*
Repeat find	.	Repite el ultimo comando *find*
Spy	s	Introduce un *spy point*
Alternatives	A	Muestra los objetivos con alternativas
Help	h	Muestra las opciones disponibles

***Figura 2.3**: Comandos depurador.*

Algunos comandos para reseñar serían el comando *spy*, que nos permite indicar un punto desde el cual podremos empezar la depuración. Esto puede resultar muy cómodo si el programa es muy largo y solo nos interesa la depuración de un punto específico. El comando A también es interesante ya que permite mostrar por pantalla los pasos de la ejecución que tienen una posible alternativa.

SWI-Prolog también dispone una interfaz gráfica para el depurador, la cual se puede ver en la figura 2.4. Sin embargo resulta bastante rudimentaria.

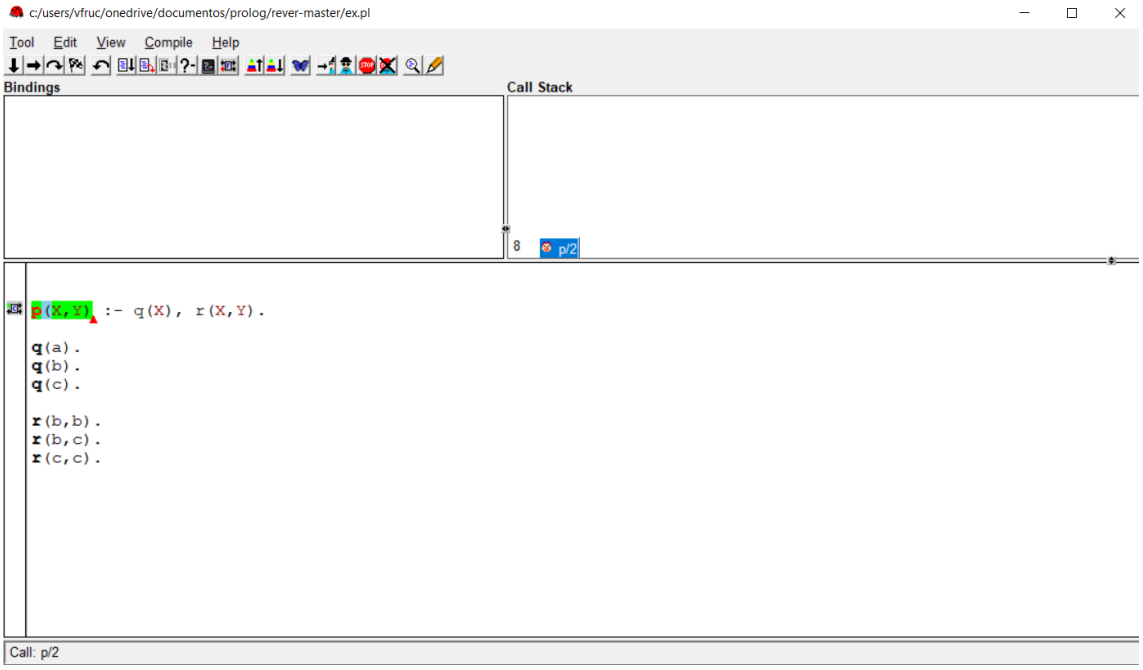


***Figura 2.4**: Interfaz depurador.*

Si bien el depurador presenta opciones interesantes, no permite volver atrás en los pasos de ejecución, lo que obliga a reejectar el programa cuando se encuentra un error inesperado. El programa rever(Vidal,G;2020) desarrollado por el tutor y que usaremos como base para este trabajo solventa este problema.

2.3 Rever

El depurador Rever(Vidal,G;2020) es el depurador que utilizaremos en este trabajo. La principal novedad de este depurador es la posibilidad de explorar una ejecución tanto hacia adelante como hacia atrás. Para esto, rever se basa en una semántica operacional reversible

El principal problema con el que se encuentra este depurador es que la unificación de Prolog es irreversible. Por lo tanto, no es tan sencillo regresar a un paso de ejecución anterior porque las variables utilizadas seguirán almacenando el valor.

En términos generales, el depurador se comportará como el depurador estándar visto arriba, mostrando los mismos pasos.

Al almacenar los pasos de ejecución en un historial, cuando queramos llevar a cabo un paso atrás podremos utilizar la información almacenada para deshacer las acciones acometidas hasta este punto.

Rever tiene dos modos de funcionamiento: El modo *debug* ejecuta el programa sin mostrar los pasos al usuario hasta alcanzar la ejecución del átomo rtrace, mientras que el modo *trace* mostrará todos los pasos de ejecución. En los siguientes capítulos mostraremos el uso del depurador Rever.

2.4 XPCE

Respecto al desarrollo de la interfaz gráfica, en un primer momento se pensó en utilizar XPCE (Anjewierden, Wielemaker;1987), la interfaz gráfica creada para SWI-Prolog la cual permite una buena integración entre el código y la interfaz ya que ambas están escritas en Prolog. Concretamente XPCE dispone de los siguientes elementos.

OBJETO	DESCRIPCION
dialog	Sirve para crear un cuadro de dialogo
button	Sirve para crear un boton
cursor	Sirve para modificar la imagen del cursor
figure	Sirve para poner imágenes en pantalla
image	Sirve para cargar imágenes
bitmap	Para convertir una imagen en un elemento gráfico (basicamente hace de puente para pasar de *image* a *figure)*
pixmap	Es practicamente equivalente a image
label	Para escribir una etiqueta por pantalla (puede servir para mostrar un texto)
menu	Para crear un menu
menu_bar	Para crear una barra de menus
menu_item	Para embeder elementos dentro de un menu
point	Para crear un punto con 2 cordenadas
popup	Para crear un popup en una ventana.
slider	Para crear un desplazador
window	Para crear una ventana donde se pueden dibujar otra serie de objetos, como imágenes, etc...

Figura 2.5: *Objetos XPCE Fuente*:
http://www.dccia.ua.es/logica/prolog/docs/ProgGUI.pdf

Sin embargo, la falta de experiencia con la herramienta y una documentación un tanto complicada de seguir hizo que se decidiera utilizar otras alternativas.

2.5 JPL/Java

JPL/Java (Dustin, Singleton, Wielemaker;2004*)* una librería de SWI-Prolog que nos permite conectar Prolog y Java. Con JPL podemos crear una interfaz gráfica utilizando Java de una manera más cómoda y flexible, mientras usamos Prolog para resolver las consultas de los predicados. El siguiente código muestra como desde Java podemos hacer una consulta Prolog para buscar una solución a una llamada:

```
Query q1 = new Query("consult('ejemplo.pl')")

 q1.hasSolution();

Query q2 = new Query("p(A,B)");

q2.hasSolution();

Map<String,Term> solution = q2.allSolutions();


 For(inti=0;i<solution.length;i++){System.out.println(
 solution[i])}
```

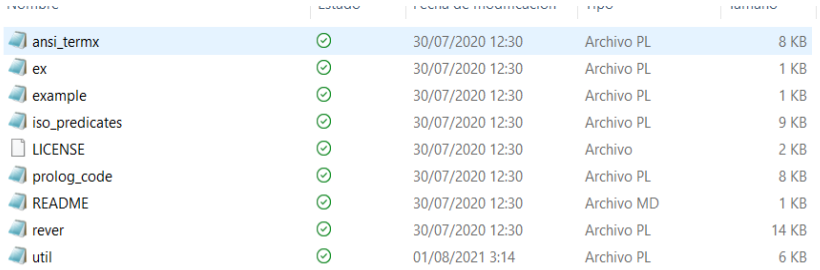
En el anterior ejemplo cargamos el archivo Prolog creando una *query* con una directiva *consult* y luego creamos una segunda para poder realizar consultas a Prolog y obtener todas las soluciones existentes.

3. Análisis del problema

Visto el estado del arte, podemos pasar a analizar los requisitos de la interfaz gráfica a desarrollar, presentando los diferentes casos de uso.

3.1 Especificaciones del programa original

Para un correcto análisis, debemos primero fijarnos en el punto de partida. En este caso, el código Prolog del depurador rever. El programa se compone de los siguientes archivos:



ansi_termx	30/07/2020 12:30	Archivo PL	8 KB
ex	30/07/2020 12:30	Archivo PL	1 KB
example	30/07/2020 12:30	Archivo PL	1 KB
iso_predicates	30/07/2020 12:30	Archivo PL	9 KB
LICENSE	30/07/2020 12:30	Archivo	2 KB
prolog_code	30/07/2020 12:30	Archivo PL	8 KB
README	30/07/2020 12:30	Archivo MD	1 KB
rever	30/07/2020 12:30	Archivo PL	14 KB
util	01/08/2021 3:14	Archivo PL	6 KB

***Figura 3.1**: Archivos programa original.*

- **ansi_termx**: Módulo provisto por SWI-Prolog para la manipulación de términos.
- **ex y example**: Programas de ejemplo.
- **iso_predicates**: Predicados destinados a la manipulación de predicados *built-in*.
- **prolog_code**: Módulo provisto por SWI-Prolog con utilidades varias.
- **rever**: Módulo principal del programa, el cual implementa la lógica del depurador.
- **util**: Utilidades dirigidas principalmente a la presentación por pantalla de resultados (*prints…*) .

Para ver su funcionamiento, utilizaremos el intérprete de SWI-Prolog, para lo cual cargaremos el ejemplo ex y el depurador. Finalmente, lo ejecutaremos llamando a rtrace(p(A,B)), como podemos observar en la siguiente imagen.

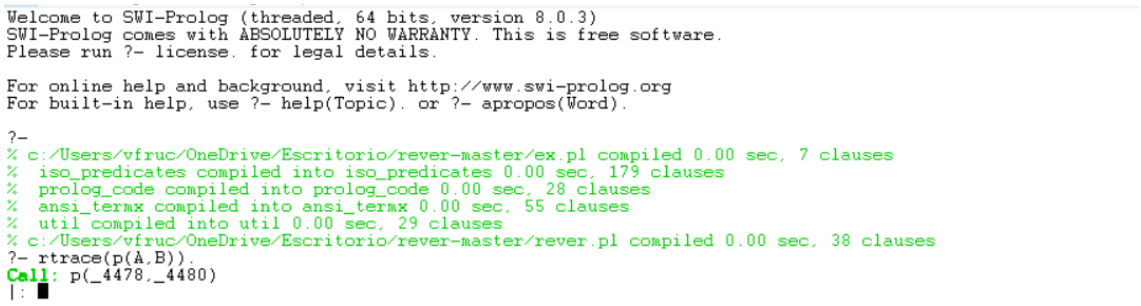


***Figura 3.2**: Ejecución programa original.*

Ahora, la aplicación nos permite realizar varias opciones según la tecla que elija el usuario:

- **Tecla *down* o *enter:*** La aplicación mostrará el nuevo estado de ejecución.
- **Tecla *up:*** La aplicación mostrará el estado anterior en el árbol de ejecución del programa.
- **Tecla *semicolon*:** Esta opción solo podrá activarse cuando se dé una primera solución. La aplicación buscará una solución alternativa.
- **Tecla s:** La aplicación se ejecutará continuamente hasta el siguiente fallo o acierto.
- **Tecla q:** Se cerrará la ejecución del depurador.

```
Welcome to SWI-Prolog (threaded, 64 bits, version 8.0.3)
SWI-Prolog comes with ABSOLUTELY NO WARRANTY. This is free software.
Please run ?- license. for legal details.

For online help and background, visit http://www.swi-prolog.org
For built-in help, use ?- help(Topic). or ?- apropos(Word).

?-
% c:/Users/vfruc/OneDrive/Escritorio/rever-master/ex.pl compiled 0.00 sec, 7 clauses
%  iso_predicates compiled into iso_predicates 0.00 sec, 179 clauses
%  prolog_code compiled into prolog_code 0.00 sec, 28 clauses
%  ansi_termx compiled into ansi_termx 0.00 sec, 55 clauses
%  util compiled into util 0.00 sec, 29 clauses
% c:/Users/vfruc/OneDrive/Escritorio/rever-master/rever.pl compiled 0.00 sec, 38 clauses
?- rtrace(p(A,B)).
Call: p(_4478,_4480)
|: Call: q(_4478)
|: Exit: q(a)
|: Call: r(a,_4480)
|: Fail: r(a,_4480)
|: Redo: q(_4478)
|: Exit: q(b)
|: Call: r(b,_4480)
|: Exit: r(b,b)
|: Exit: p(b,b)
|: **Solution: [p(b,b)]
|: Redo: r(b,_4480)
|: Exit: r(b,c)
|: Exit: p(b,c)
|: **Solution: [p(b,c)]
|: Redo: q(_4478)
|: Exit: q(c)
|: Call: r(c,_4480)
|: Exit: r(c,c)
|: Exit: p(c,c)
|: **Solution: [p(c,c)]
No more solutions...
|:
```

***Figura 3.3**: Ejecución completa programa original.*

Como podemos ver en el ejemplo de la imagen anterior, la aplicación mostrará la ejecución paso a paso. Con esta información podemos determinar los requerimientos básicos que necesitará la interfaz gráfica.

3.2 Requisitos del sistema

3.2.1 Propósito

El presente apartado tiene como objetivo la presentación de los requisitos de la aplicación a desarrollar.

3.2.2 Ámbito del sistema

El proyecto deberá implementar una aplicación que permita a los usuarios la visualización de la ejecución de un programa Prolog paso a paso, observando todos sus pasos de ejecución.

La interfaz debe ser simple y orientada a su uso por parte de alumnado poco familiarizado con el uso del lenguaje Prolog. Esta aplicación será muy beneficiosa en el aprendizaje y la utilización de este lenguaje ya que presentará a los usuarios una visión completa de sus pautas de funcionamiento, lo cual ayudará a una mayor comprensión. El objetivo final es la utilización de la aplicación en diversos ámbitos educativos para ayudar en el aprendizaje de este lenguaje de programación.

3.2.3 Definiciones

Java: Lenguaje de programación dirigido a objetos.

Prolog: Lenguaje de programación lógico.

Pasos de ejecución: Partes en las que se divide el trabajo realizado por una aplicación para resolver un problema.

Solución: Situación en la que un programa Prolog da una respuesta a una llamada efectuada por el usuario.

Árbol de ejecución: Representación de todos los caminos posibles que puede tomar un programa Prolog para intentar llegar a una solución.

3.2.4 Referencias

Especificación de requisitos según el estándar de IEEE 830,2008 https://www.fdi.ucm.es/profesor/gmendez/docs/is0809/ieee830.pdf

3.2.5 Perspectiva del producto

La aplicación es independiente de otros productos existentes. La única dependencia que presenta es la necesidad de que el ordenador que ejecute la aplicación pueda ejecutar programas Java y SWI-Prolog. Si esto se cumple,la aplicación es independiente de cualquier otro sistema.

3.2.6 Funciones del producto

El proyecto deberá permitir a los usuarios realizar las acciones de:

- Carga de programas Prolog.
- Mostrar nuevo paso de ejecución.
- Regresar al anterior estado de ejecución.
- Obtención de soluciones alternativas.

3.2.7 Características de los usuarios

Como hemos explicado anteriormente los usuarios serán usualmente estudiantes de informática o de otras disciplinas que requieran del aprendizaje del lenguaje Prolog, poco versados en el manejo de este.

También se pueden incluir los usuarios con conocimientos informáticos y con un nivel medio o alto en Prolog que quieran utilizar la aplicación para otros fines.

3.2.8 Suposiciones y Dependencias

Se asume que ninguna alteración externa referente a los lenguajes aquí utilizados cambie de una manera que afecte al correcto funcionamiento del programa.

3.2.9 Requisitos futuros

En el futuro se podría plantear la función de obtención de los pasos de ejecución de una manera más visual, por ejemplo, mostrando la ejecución en una estructura de árbol.

3.2.10 Requisitos de interfaces externas

La aplicación no necesita interactuar en ningún momento con alguna aplicación ya existente.

3.2.11 Requisitos funcionales

Id	Re1
Nombre	Nuevo paso ejecución
Descripción	El usuario podrá pedir que se ejecute un nuevo paso del programa
Entrada	Programa + estado actual
Salida	Vista del paso de ejecución
Prioridad	Alta

Id	Re2
Nombre	Llamada paso anterior
Descripción	El usuario podrá pedir que se vuelva al paso anterior de ejecución.
Entrada	Programa + estado actual
Salida	Vista de anterior paso de ejecución
Prioridad	Alta

Id	Re3
Nombre	Introducción llamada
Descripción	El usuario podrá introducir la llamada a la a que quiere encontrar solución.
Entrada	Objetivo(query)
Salida	Introducción de la llamada en el programa
Prioridad	Alta

Id	Re4
Nombre	Carga de archivo Prolog
Descripción	El usuario podrá cargar el archivo que contendrá las instrucciones del programa.
Entrada	Fichero .pl
Salida	Carga de archivo en la aplicación
Prioridad	Alta

Id	Re5
Nombre	Abrir información
Descripción	El usuario podrá recibir información sobre el creador de la aplicación.
Entrada	-
Salida	Vista de ventana de información

Prioridad	Baja

3.2.12 Requisitos de rendimiento

La aplicación debe cargar los archivos y efectuar todas sus acciones de modo que los tiempos de carga sean imperceptibles por el usuario.

4. Diseño de la solución

En este apartado presentaremos el desarrollo de la solución explicando las diferentes partes en las que subdividiremos el programa y cómo las implantaremos en la solución.

4.1 Metodología

La metodología que hemos aplicado en el diseño es la metodología RUP (Booch,Rumbaugh,Jacobson;1998) es decir hemos dividido el trabajo en 4 fases:

- Inicio, donde hemos articulado las primeras ideas.
- Elaboración, donde hemos fijado los casos de uso.
- Desarrollo, donde hemos construido las funcionalidades del programa.
- Transición, donde comprobaremos el correcto funcionamiento de las funcionalidades.

4.2 Casos de uso

Ahora que ya hemos explicado las especificaciones iniciales, podemos pasar a explicar los casos de uso. El primero que se planteó fue el siguiente: Un usuario debe poder hacer cinco acciones: Carga del programa, introducción de la llamada a ejecutar, obtención de un nuevo paso de ejecución, obtención del paso de ejecución anterior y obtención de una solución alternativa.

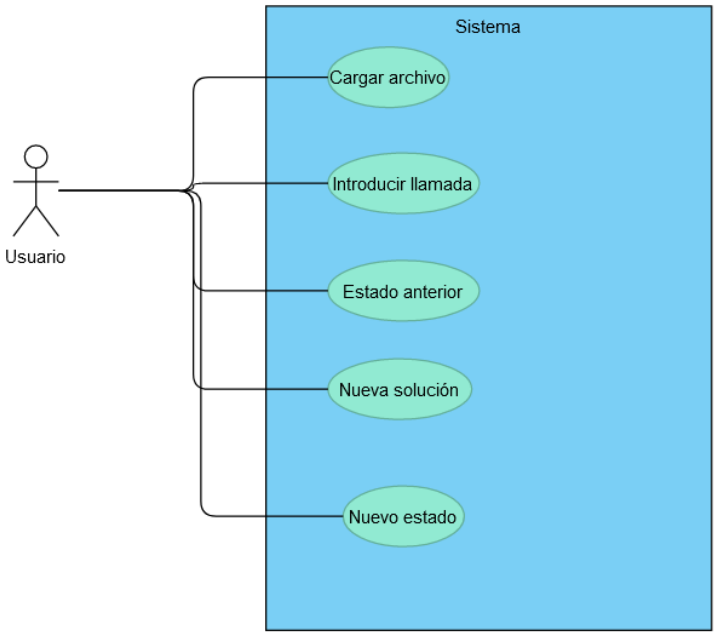


***Figura 4.1**: Primer caso de uso.*

Sin embargo, en el momento de crear el *mockup* de la aplicación se decidió que un botón único para la llamada a una nueva solución resultaba tedioso y se decidió que básicamente como ésta era otra llamada a un nuevo estado, se agruparían estas dos acciones en una sola. Además, se incluyó una opción para obtener información acerca de la aplicación. El nuevo caso de uso queda de la siguiente manera:

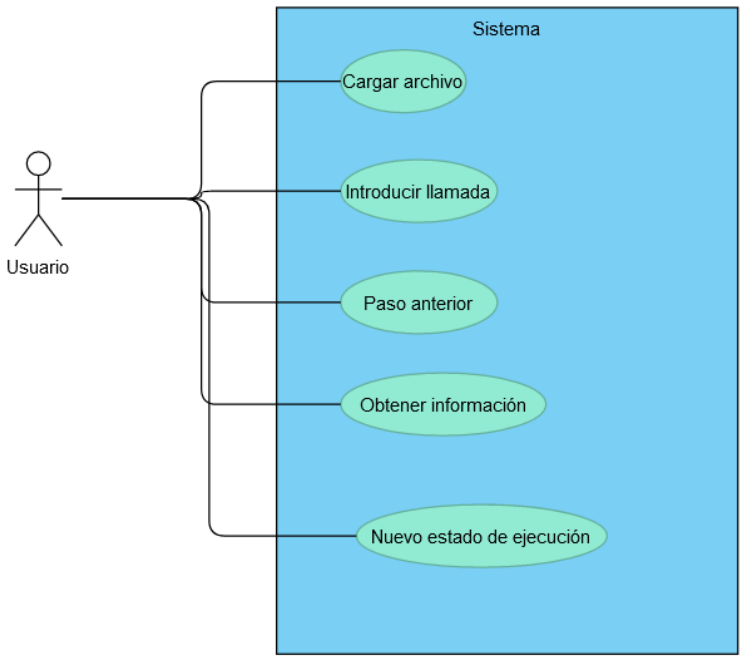


***Figura 4.2**: Segundo caso de uso.*

4.3 Mockup y diseño de la aplicación

Después de diseñar los casos de uso, se pasó al diseño de la interfaz gráfica que debe integrar todas estas acciones. La principal idea de diseño fue la búsqueda de una interfaz simple y sencilla, ya que está destinada al aprendizaje. Finalmente se decidió por la distribución ilustrada En el siguiente *mockup*:

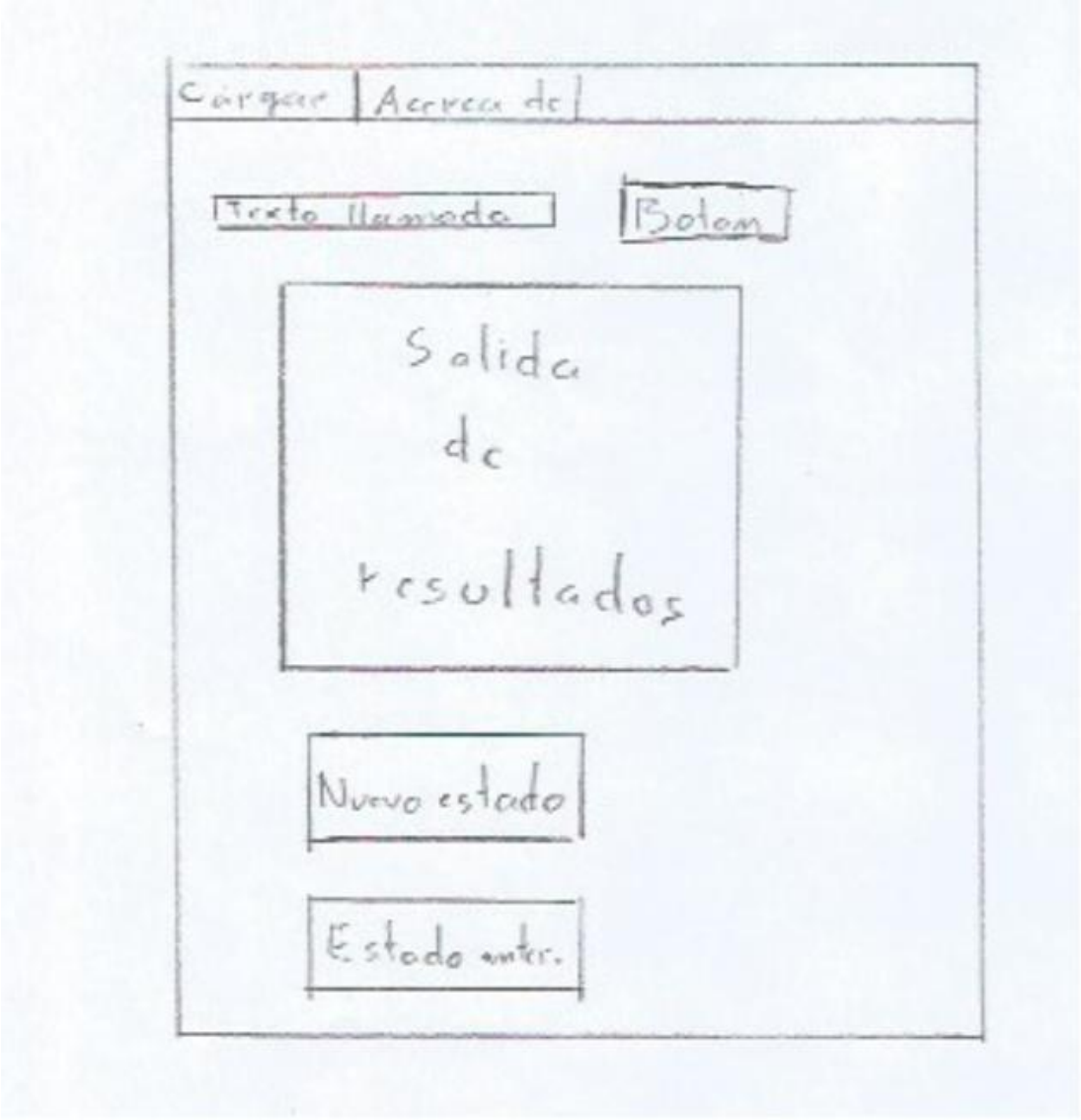


***Figura 4.3**: Mockup de la interfaz.*

En la parte superior de la aplicación tendremos una barra de menú que almacenará las opciones de cargar archivos y la de información sobre la autoría de la aplicación. Al pulsar el botón de carga, se nos abrirá un submenú que nos dejará cargar el archivo con el programa a depurar. Si pulsamos el botón de "Acerca de" nos saltará una pestaña con datos como el nombre del autor de la aplicación, tutor del TFG, universidad del creador y curso en el que se creó.

Debajo de este menú encontraremos una *Textbox* o caja de texto donde el usuario deberá introducir la inicial, y a su derecha, un botón que ejecutará dicha llamada.

Para mostrar los resultados de las acciones del programa utilizaremos un panel de texto donde iremos añadiendo los resultados. Es importante que éste sea retráctil ya que alguna llamada con varias soluciones podría ocupar la totalidad del panel y, en ese caso, el usuario debe poder seguir teniendo la posibilidad de ver todos los resultados.

Para las opciones de nuevo estado y de recuperar el estado anterior utilizaremos el mismo método con ambos: Un botón que al pulsarlo interactuará con el panel antes mencionado y mostrará el nuevo estado o lo eliminará, respectivamente.

4.4 Soluciones alternativas

Como hemos explicado antes, el programa se podría haber dejado con la opción de llamar a una nueva solución como una opción a cargo de un botón independiente, pero al comenzar la fase de programación nos dimos cuenta de que el usuario que buscara consultar alguna de las soluciones alternativas encontraría muy engorroso el ir cambiando.

Otra versión que se planteó es que en vez de una presentación de los datos en forma de lista, la aplicación los mostrara en su forma original: un árbol de ejecución en el que pudiéramos ver claramente los distintos caminos que el programa puede tomar. Si bien esta opción es mucho más vistosa, aumentaba mucho la complejidad de la interfaz gráfica.

4.5 Arquitectura del sistema

En lo que se refiere a la arquitectura se decidió que el primer nivel se compondría por las cinco acciones que todo usuario puede hacer en la aplicación: Carga de archivos, nuevo estado, estado anterior, introducción de llamada e información sobre la aplicación, como representa la siguiente figura.

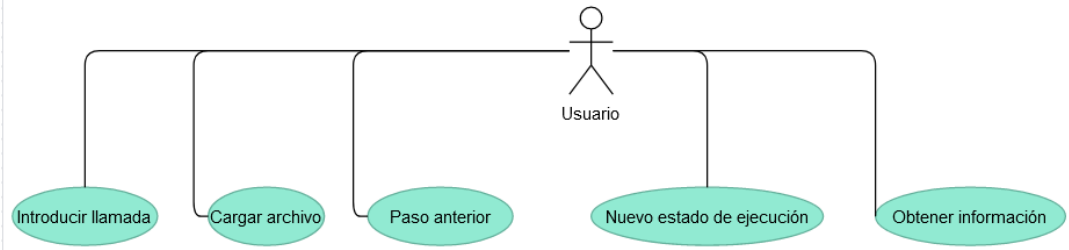


***Figura 4.4**: Primer nivel de arquitectura.*

Ahora vamos a hacer hincapié en la instrucción de obtener nuevo estado. En lo referente a estas otras llamadas y tomando como base el código original podemos ver en la figura 4.5 lo que hemos comentado en el capítulo del estado del arte sobre el modelo que siguen las salidas, usando *call/exit/fail y redo*.

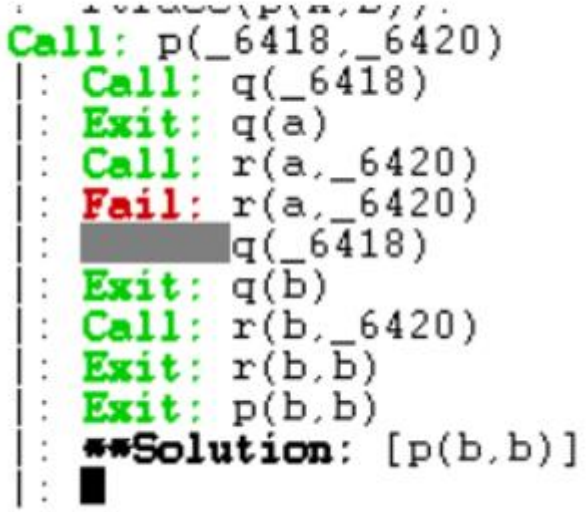


***Figura 4.5**: Ejemplo de posibles salidas.*

En el caso de la recuperación del estado anterior, el programa debería comportarse de la misma manera sea cual sea el estado que se encuentre. El nuevo caso de uso queda de la siguiente forma:

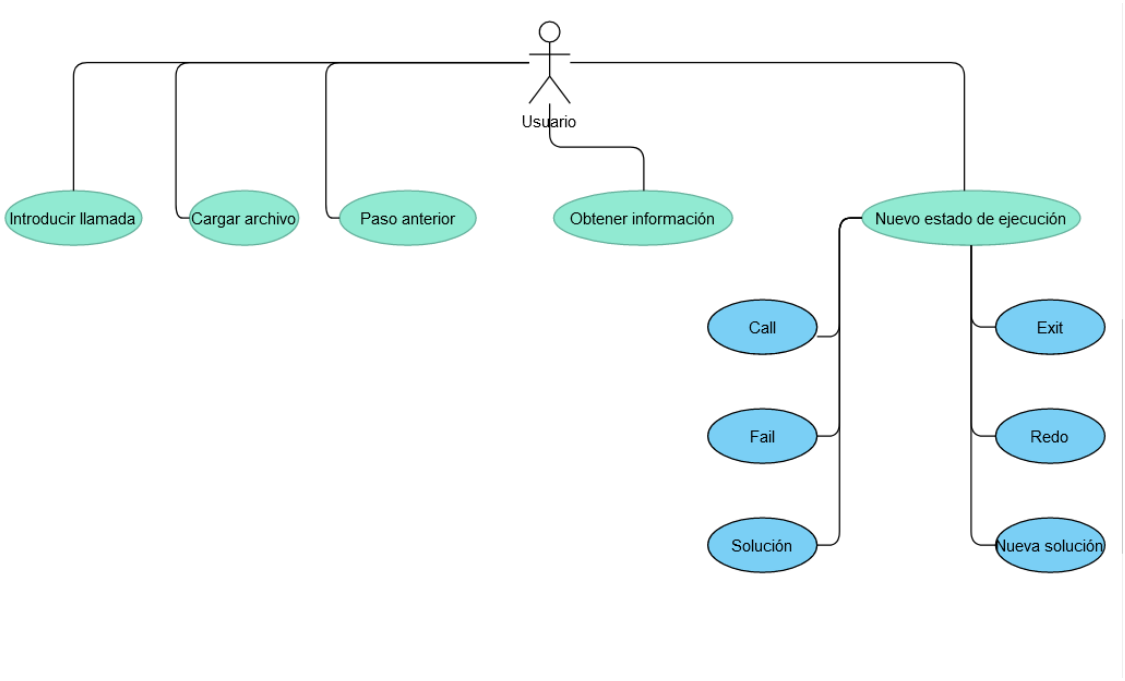


***Figura 4.6**: Segundo nivel de arquitectura.*

4.6 Diseño detallado

Recordando lo estipulado en el trabajo, utilizaremos Java desarrollar la interfaz gráfica manteniendo así el programa rever prácticamente sin cambios. Esto se refleja en la siguiente figura:

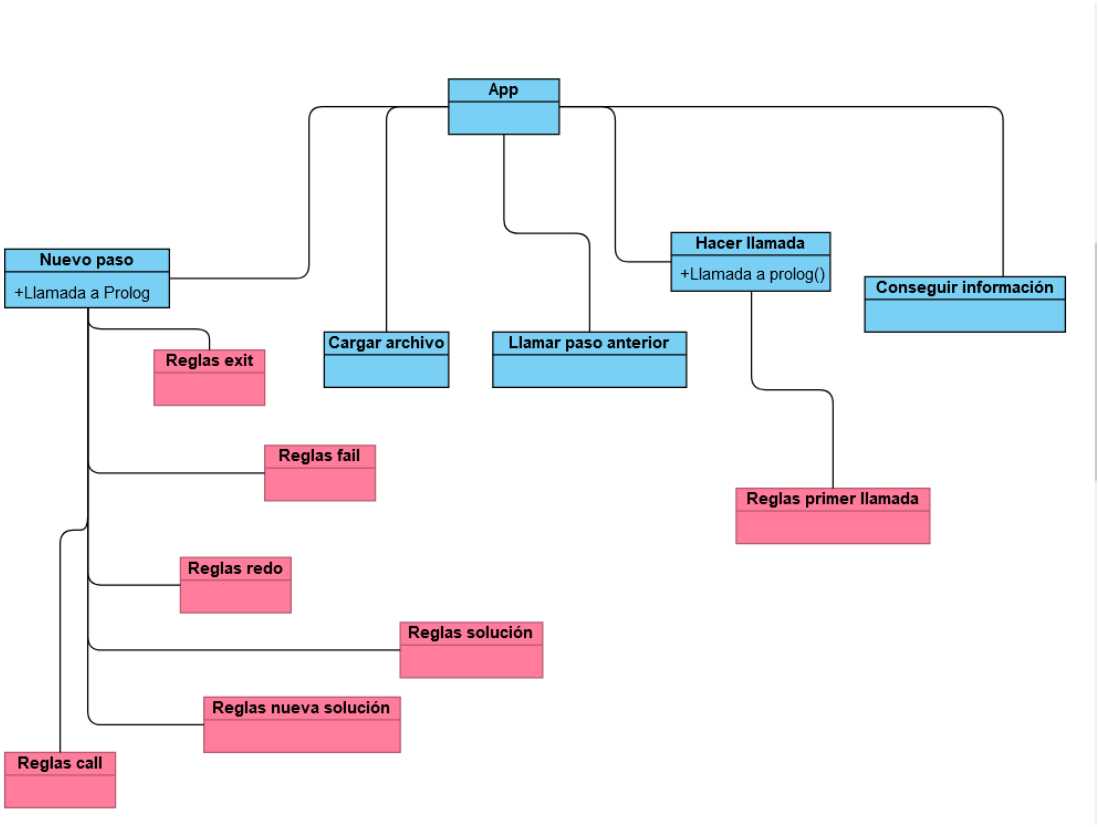


***Figura 4.7**: Diagrama de la aplicación. En rojo las partes hechas en Prolog y en azul las partes en Java.*

4.7 Tecnologías utilizadas

SWI-Prolog: Versión en código abierto del lenguaje Prolog cuyo objetivo es la escalabilidad, pudiendo soportar múltiples procesos aprovechando el hardware de las máquinas, ofreciendo también una gran cantidad de herramientas de desarrollo. La versión utilizada en el proyecto es la 8.0.3

***Figura 4.8**: Logo SWI-Prolog.*

JPL/Java: Librería de SWI-Prolog que permite una conexión bidireccional entre los lenguajes Java y Prolog. JPL permite que las aplicaciones en Prolog puedan aprovechar las clases, métodos, etc. que puede aportar Java a la vez que las propias aplicaciones en Java pueden utilizar las librerías de Prolog.

JDK Java: Kit de desarrollo de Java que incluye todas las herramientas necesarias para una correcta utilización de este lenguaje: intérprete, herramientas de desarrollo… Debido que la versión actual del JDK es incompatible con la versión de SWI-Prolog ha sido necesario utilizar una versión anterior, en este caso la JDK 8.

Java: Librería de Java que contiene las herramientas gráficas del lenguaje.

Visual Studio Code: Herramienta de escritura de código gratuita de Microsoft que ofrece un gran abanico de opciones para múltiples lenguajes, además de acomodar la escritura a los usuarios con una amplia gama de opciones y ayuda para la escritura de programas.

***Figura 4.9**: Logo Visual Studio Code.*

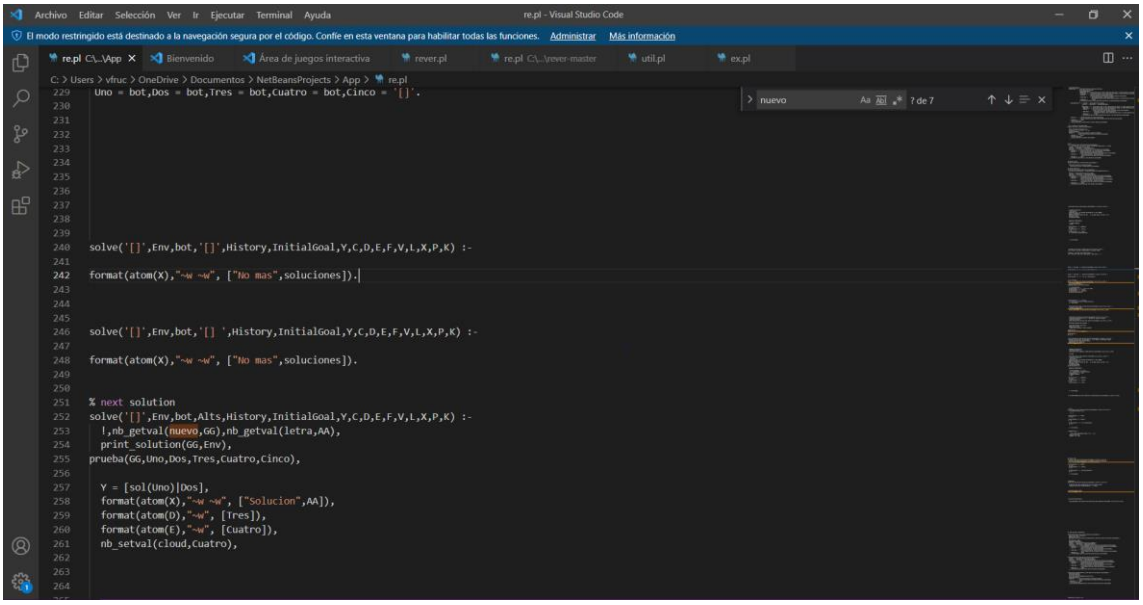


***Figura 4.10**: Interfaz Visual Studio Code.*

NetBeans: Herramienta de desarrollo de trabajos en lenguaje Java, NetBeans permite tanto la escritura de código como la creación de interfaces gráficas todo de una manera cómoda y permitiendo al usuario un mejor control de sus proyectos. Para buscar una mejor interacción con la versión del JDK, además de por gusto personal, hemos utilizado una versión más antigua de Netbeans; específicamente, la 8.2.

***Figura 4.11**: Logo NetBeans versión 8.*

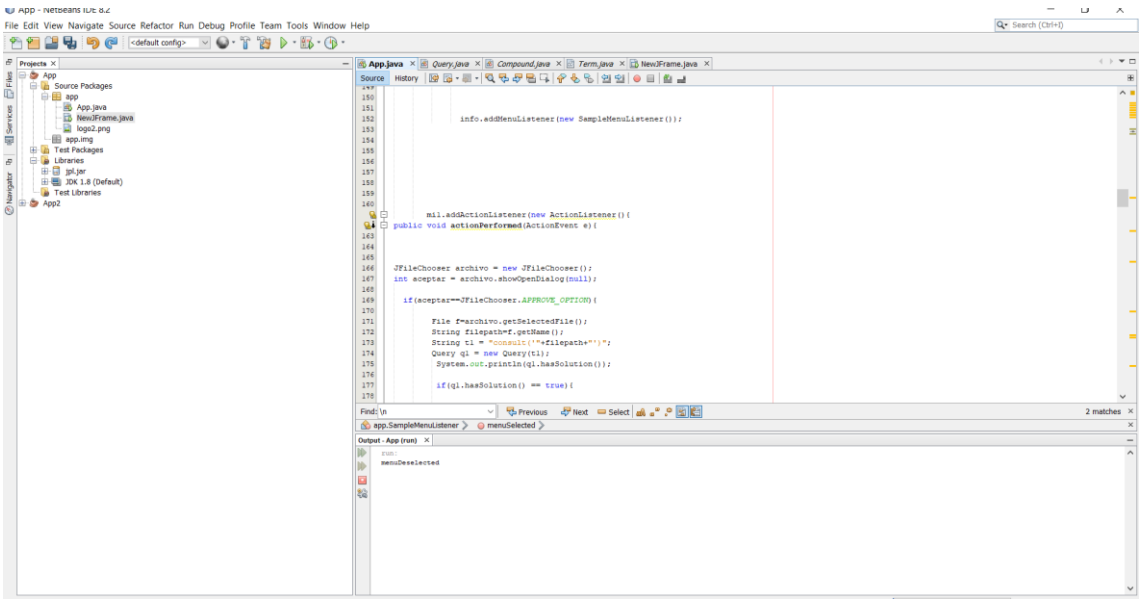


***Figura 4.12**: Interfaz NetBeans versión 8.*

5. Desarrollo de la solución

En este apartado vamos a presentar los detalles de la implementación final de la aplicación. Siempre que podamos, seguiremos la misma pauta: Presentaremos la parte escrita en Java para después explicar la parte asociada al código Prolog.

En términos generales hemos implementado una interfaz que permitirá cargar un archivo seleccionado por el usuario y, después permitirá, que éste introduzca una llamada al programa. Luego, mostrará por pantalla las órdenes del usuario según el botón que éste accione.

Para comenzar vamos a crear un JFrame que es básicamente una ventana desde la que podremos almacenar todos los demás objetos que compondrán la aplicación. En este momento, añadimos los dos primeros componentes: Una JTextArea, en la que escribiremos los resultados obtenidos para mostrárselos al usuario, y un botón para que el usuario pueda obtener un nuevo estado. Es importante que la TextArea esté imbuida en una ScrollArea, un componente que permite que ésta al llenarse siga almacenando nuevos caracteres más allá de su límite. Esto es importante ya que algunas trazas de ejecución podrían ser bastante extensas

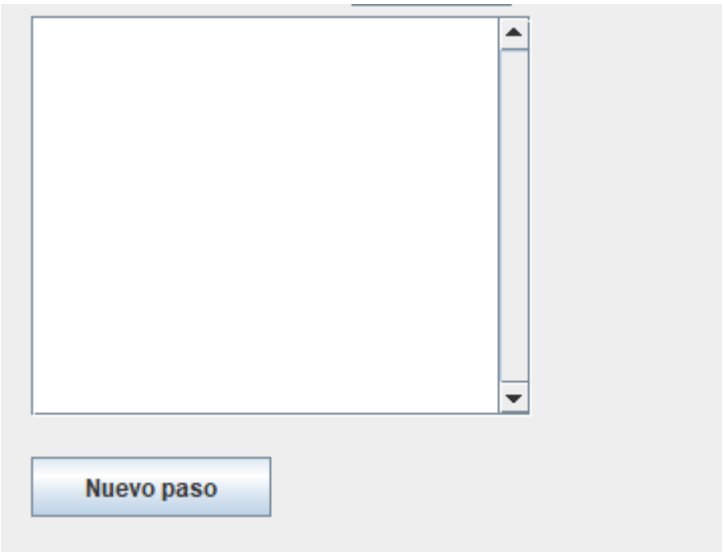


***Figura 5.1**: TextArea y botón.*

También cargaremos directamente el programa rever y la primera llamada de cara a implementarlas después. Para hacer esto, seguiremos la siguiente estructura que, si bien en este caso es temporal, se usará repetidamente para conectar la aplicación con el código Prolog.

```
String t2 = "consult('re.pl')";

Query q2 = new Query(t2);

System.out.println(q2.hasSolution());
```

Aquí podemos ver como JPL nos ayuda a interconectar ambos lenguajes; lo que vamos a hacer es crear un *string*, el cual será la consulta que queremos hacer, ya sea una llamada o un archivo .pl. Esto lo pasaremos luego a una *query* donde comprobaremos si ésta tiene solución.

Hechos estos primeros pasos, podemos comenzar con la primera tarea que será la de la obtención de nuevo paso de ejecución.

5.1 Nuevo paso

Antes de comenzar la explicación de la parte de Java, conviene comenzar presentando cual será una llamada típica en el depurador rever.

```
solve([A|T],Env,bot,Alts,History,InitialGoal,Y,C,D,E,
F,V,L,X,P,K) :-
```

Si bien cada caso puede verse modificado para adecuarse a la situación, la estructura es siempre prácticamente la misma. Los parámetros toman los siguientes valores:

- [A|T]: Objetivo actual.
- Env: Almacena el valor de cada variable.
- Bot o Cl: Etiqueta de la cláusula. Este apartado tomará dos tipos de valores: La variable CL que almacenará el valor de la etiqueta de una consulta para poder diferenciarla de las demás o el valor bot si no es necesaria ninguna identificación.

- History: Lista de los estados por los que hemos pasado hasta el momento de la llamada.
- InitialGoal: Almacena la llamada inicial al programa.
- Variables: Se utilizarán para almacenar la información de los campos anteriores después de terminar un paso de ejecución para poder mandársela a la interfaz Java y que pueda llamar a la siguiente orden.

Explicados estos conceptos, podemos comenzar con el desarrollo. Como hemos explicado antes, debemos primero preparar una *query* que servirá como llamada al depurador. En este caso, seguiremos el mismo plan con un añadido.

```
String t4 = "solve("+YY+ ","+DD.name()
+","+EE+","+FF+","+GG+","+HH+","+"Y,C,
D,E,F,V,L,X,P,K"+")";

System.out.println(t4);

Map<String,Term> solution2;

Query q4 = new Query(t4);

System.out.println(q4.hasSolution());

solution2 = q4.oneSolution();

System.out.println("Call:"+ solution2.get("X") + ".");

text.append(solution2.get("X") + "."+"\n");
```

Como se puede ver en este caso cargamos en el *string* la llamada que queremos consultar; además, ahora tenemos un *map* que almacenará las soluciones. Si uno se fija podrá ver que pasamos variables en la llamada. Eso se debe a que en el momento de acabar un paso de ejecución los resultados obtenidos se almacenarán en estas variables que se usarán para poder llamar al siguiente paso de ejecución.

Como hemos explicado anteriormente en el trabajo, una llamada solve puede accionar un evento call, exit redo o fail. Cada evento tiene un predicado solve distinto y según que valor tomen los argumentos en el momento de la llamada al paso de ejecución ejecutarán un evento u otro.

5.2 Hacer primera llamada

Este paso hace referencia a la creación del mecanismo que permitirá al usuario la introducción de la llamada cuya ejecución quiere explorar. Como se indica en la figura 5.2, utilizaremos una TextBox donde el usuario escribirá la llamada y un botón que al pulsar iniciará su ejecución el sistema.

***Figura 5.2**: TextBox y botón.*

Ahora, guardamos en un *string* el texto que el usuario haya introducido que servirá de base para la llamada a rtrace con la que iniciaremos la depuración. Además, para dejar claro que la carga se ha producido y que el usuario lo sepa, hacemos que se muestre por pantalla la entrada introducida.

Además, vamos a prepararnos para los casos en los que o bien el usuario no escribe nada o bien escribe algo que no es compatible como la cadena “1234”. En el primer caso ya la propia acción de pulsar el botón nos devolverá una *PrologException*, así que usaremos una estructura try-catch para atrapar la excepción y mostrar un mensaje de error.

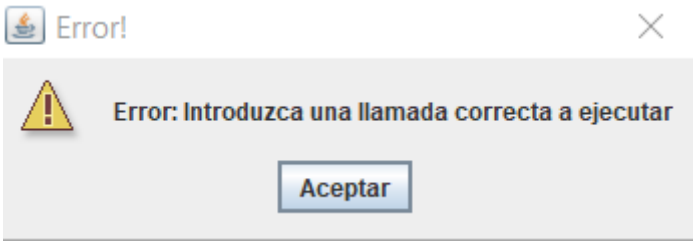


***Figura 5.3**: Mensaje error.*

En el caso de que se introduzca una llamada errónea, el programa no puede distinguir a primeras si se trata de algo erróneo, por lo que permitirá que pase a la parte Prolog. Sin embargo, al no poder hacer *matching* con ninguna cláusula saltará la regla de fail lo que provocará una *nullpointerexception.* En el caso de que sea un carácter que no comprende, no llegará a saltar la instrucción fail y saltará una *PrologException,* las cuales cazaremos con el mismo mecanismo en ambos casos, mostraremos por pantalla un mensaje indicando que no hay solución. Las figuras 5.4 y 5.5 muestran las siguientes situaciones:

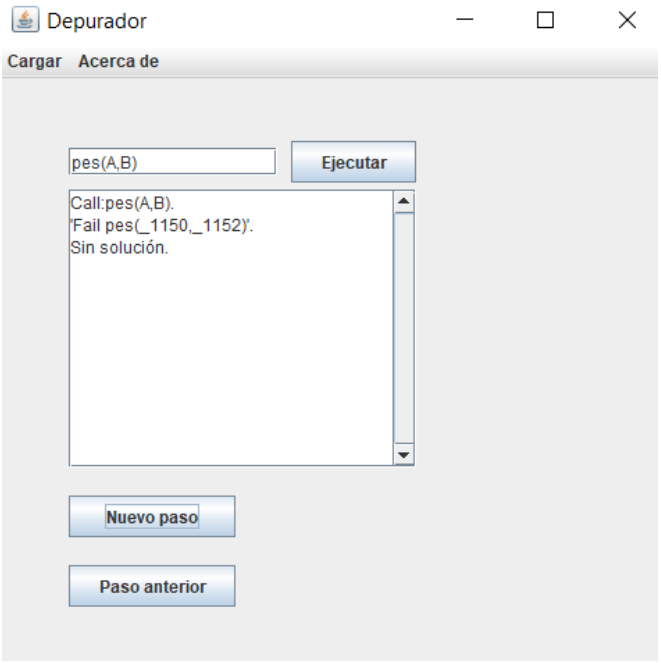


***Figura 5.4**: Ejemplo NullPointerException.*

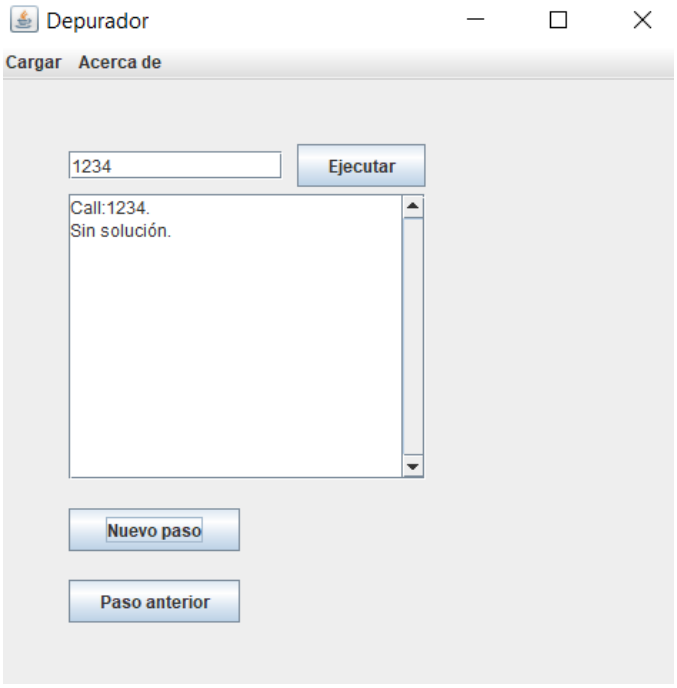


***Figura 5.5**: Ejemplo PrologException.*

5.3 Mostrar estado anterior

Este paso podría hacerse de una manera parecida que el caso de llamada a nuevo paso; sin embargo, esto podría crear problemas entre llamadas y pérdidas de información, así que utilizaremos una aproximación más sencilla, utilizando solo la parte de Java para completarla.

Para ello, vamos a seguir la siguiente estrategia: vamos a modificar el código de nuevo paso de ejecución para guardar en un array los resultados de cada una de las variables que componen las llamadas a nuevo paso. Luego este array de resultados lo almacenaremos en una lista para así ordenar estas colecciones según su orden de ejecución y poder recuperarlos más cómodamente. También crearemos dos contadores para calcular la posición en la lista en la que nos encontramos y la posición en la TextArea.

En el evento que controla esta instrucción recogemos el elemento de la lista que nos marca el contador y guardamos en las variables utilizadas para las llamadas de las *query* solve los valores guardados en el array recuperado. Luego, eliminamos el array de la lista y borramos el valor de la TextArea siguiendo los límites del contador para finalmente modificar el valor de los contadores. Por poner un ejemplo si pedimos volver

al primer paso de ejecución después de haber ejecutado cuatro más pulsaremos el botón de estado anterior que eliminará los estados ejecutados hasta llegar al primero. Además este método habrá dejado preparadas las variables usadas en solve para que cuando el usuario quiera obtener un nuevo paso de ejecución obtenga la llamada correcta que iría después de esta primera. También ponemos una condicional que impida que el programa vaya más allá de la primera entrada.

5.4 Carga archivos

Para cargar los archivos utilizaremos una utilidad aportada por el propio javax: JFileChooser, el cual nos permite seleccionar un archivo la ruta de su localización, la cual usaremos para pasársela a una directiva consult que nos permitirá cargar el archivo seleccionado. Si esto funciona, mandaremos un mensaje al usuario de que la carga se ha realizado. Visualmente, crearemos una barra de menú donde añadiremos la opción de cargar donde el usuario podrá elegir qué archivo quiere cargar. Podemos ver el resultado en la figura 5.6.

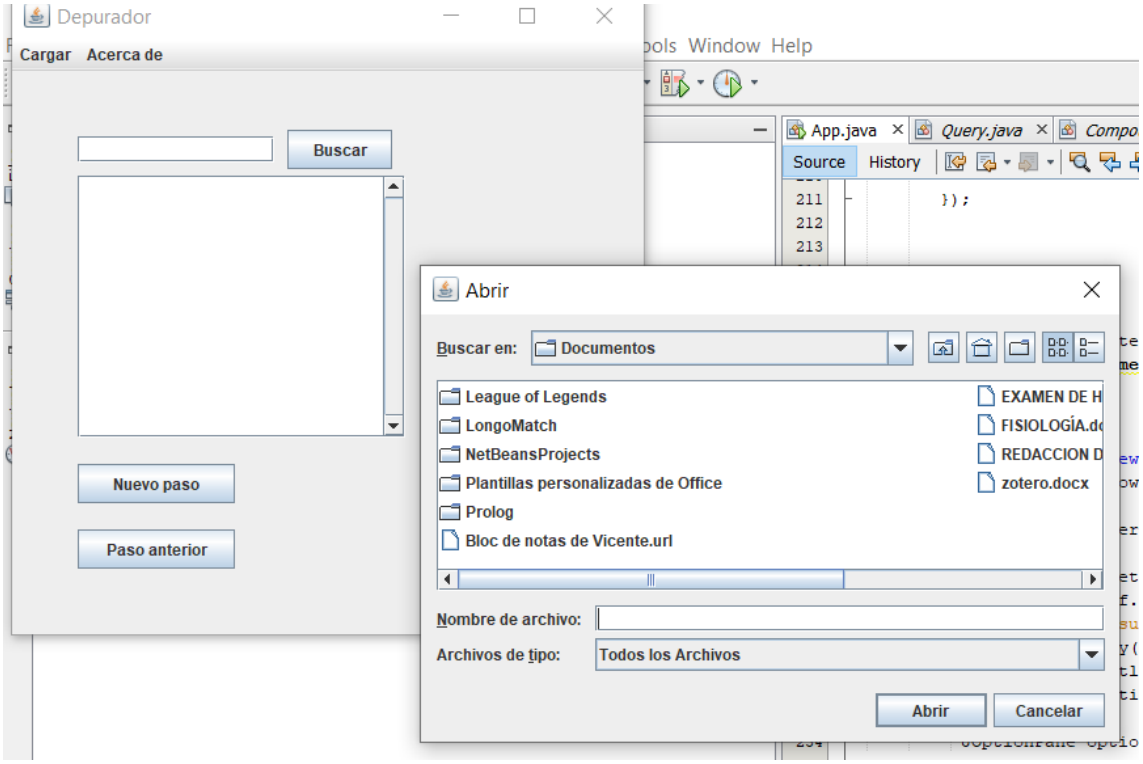


***Figura 5.6**: Interfaz carga.*

5.5 Conseguir información

Para crear la pestaña que almacenará la información sobre el trabajo, utilizaremos una utilidad del propio NetBeans que nos permite crear interfaces. Para ello, crearemos en el proyecto una nueva ventana que utilizaremos para introducir la información como se muestra en la figura 5.7.

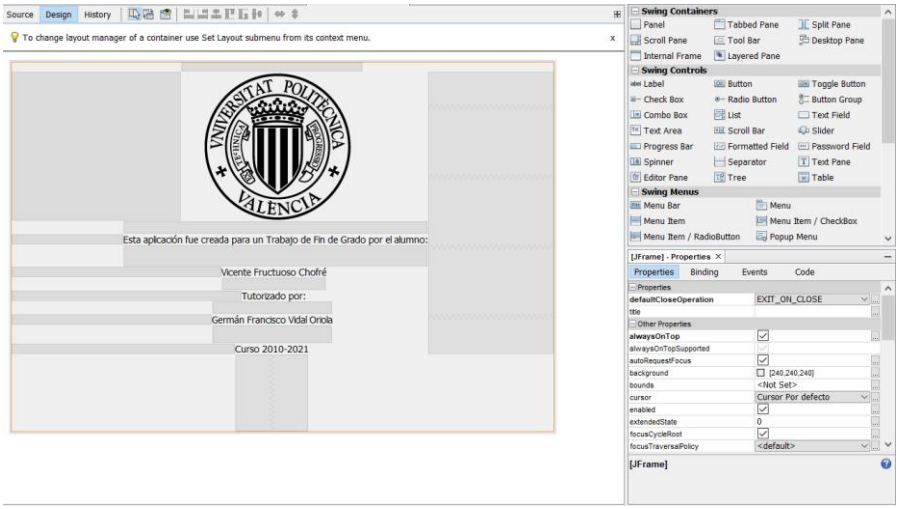


***Figura 5.7**: Diseño información.*

Para poder abrir la ventana, añadiremos una opción más a la barra de menú de carga de documentos, en la cual crearemos un MenuListener que nos permitirá controlar la apertura de ventanas.

6. Implantación y pruebas

En este apartado presentaremos la implantación de la aplicación, para lo cual usaremos el siguiente ejemplo para probar la aplicación.

```
p(X,Y) :- q(X), r(X,Y).

q(a).
q(b).
q(c).

r(b,b).
r(b,c).
r(c,c).
```

***Figura 6.1**: Ejemplo.*

De primeras vamos a hablar de la carga de archivos. Concretamente la aplicación devuelve un mensaje de carga correcta como el siguiente.

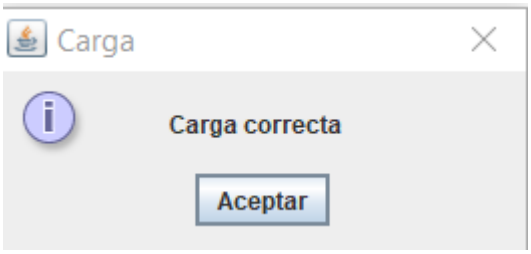


***Figura 6.2**: Feedback carga.*

Vamos ahora a intentar explorar la ejecución de la llamada p (A, B), para ello introduciremos la llamada p(A,B) en el textarea y pulsaremos el botón de ejecutar.

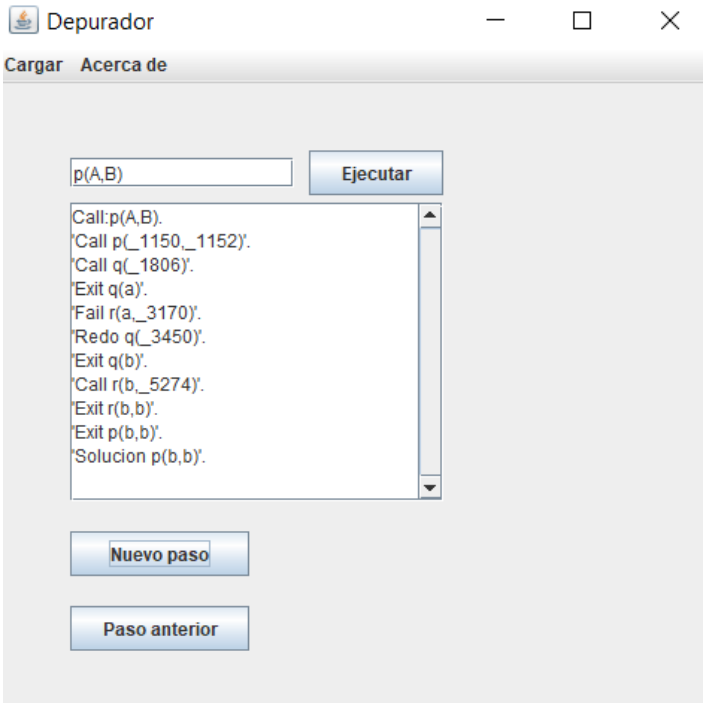


***Figura 6.3**: Ejemplo primera solución.*

Como podemos observar de la llamada p (A, B) obtenemos el resultado p(b,b), lo cual tiene sentido ya que nos piden que cumplamos q(A),r(A,B). Podemos ver cómo se efectúan las diversas llamadas de *call*, *exit*, *fail*, *redo* y la salida de soluciones. Ahora vamos a buscar soluciones alternativas. Obtenemos algo como esto:

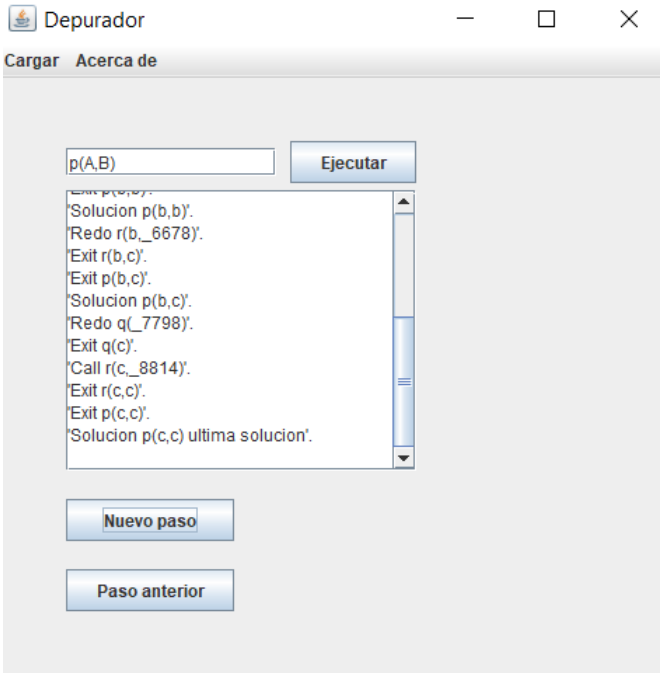


***Figura 6.4**: Ejemplo varias soluciones.*

Vamos a probar ahora con una llamada que produzca un fallo, como podría ser p(d,B. Para ello introduciremos la llamada cuyo resultado se puede ver en la figura 6.5.

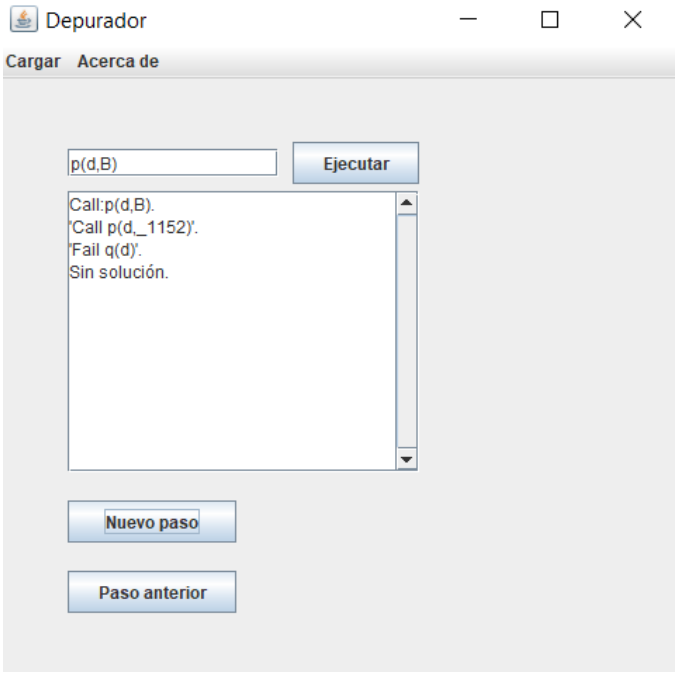


***Figura 6.5**: Ejemplo fallo.*

Para probar la opción de estado anterior vamos a ejecutar una llamada hasta un punto concreto, como se muestra en la figura 6.6.

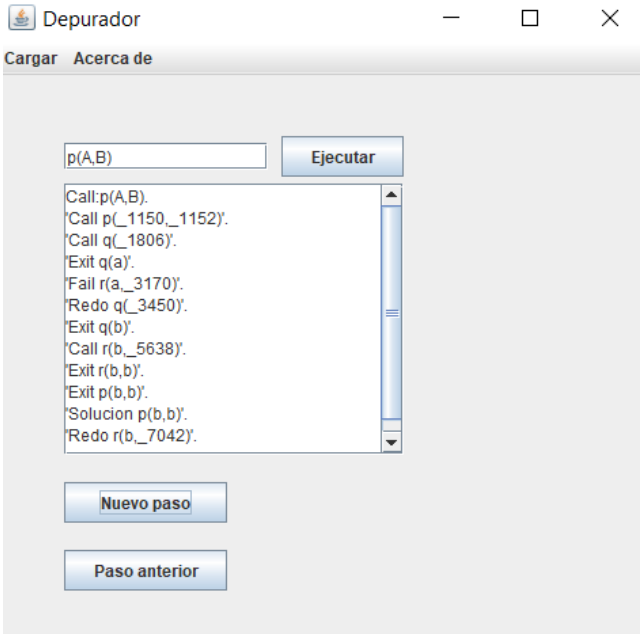


***Figura 6.6**: Ejemplo paso anterior 1.*

Ahora, como se muestra en la figura 6.7, ejecutamos estado anterior para volver al punto del primer redo, eliminando los pasos mostrados hasta ese punto.

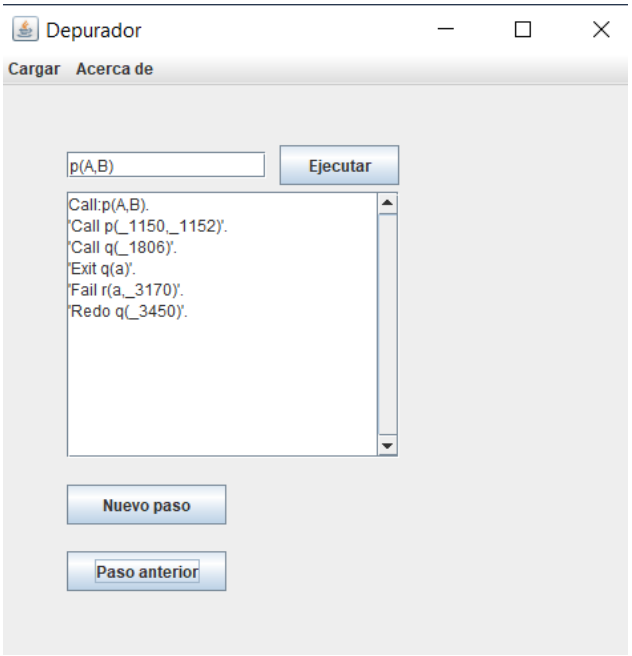


***Figura 6.7**: Ejemplo paso anterior 2.*

Y finalmente volvemos a pulsar nuevo paso hasta volver al paso en que nos encontrábamos, viendo que la ejecución es correcta

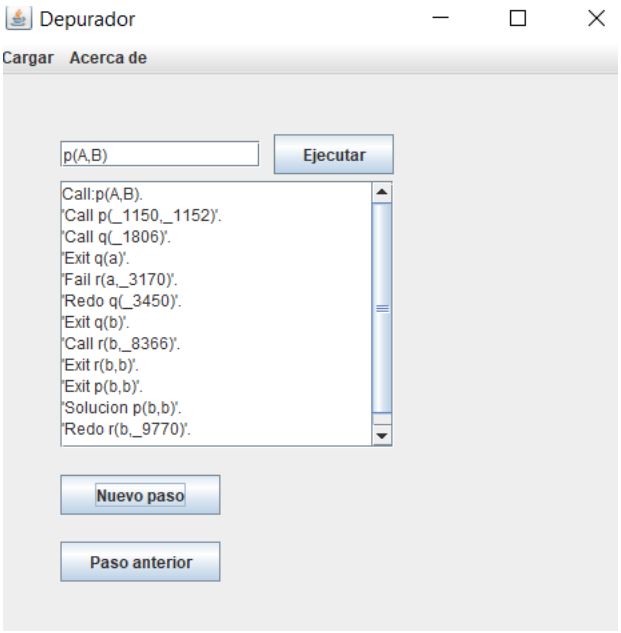


***Figura 6.8**: Ejemplo paso anterior 3.*

Para acabar podemos probar a abrir la opción de información que aparecerá como en la figura 6.9.

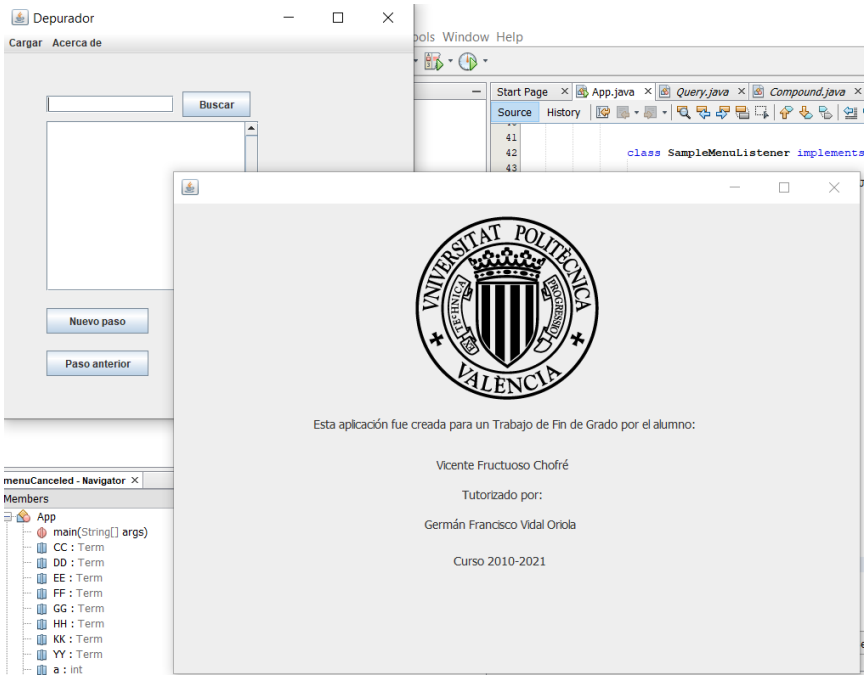


***Figura 6.9**: Ejemplo información.*

7. Conclusiones

En conclusión, hemos conseguido cumplir con los objetivos planteados

Pese a que el dominio del autor con alguno de los lenguajes empleados no era elevado y desconocía el uso de algunas herramientas, se ha conseguido un aprendizaje de estas que le ha permitido la obtención de los objetivos planteados.

7.1 Relación del trabajo desarrollado con los estudios cursados

En este trabajo hemos explotado los conocimientos adquiridos sobre programación a lo largo de la carrera.

También hemos puesto en práctica los conocimientos aprendidos en diseño de interfaces gráficas para intentar crear una interfaz sencilla y cómoda para el usuario.

El manejo de herramientas como NetBeans o Visual en diversas asignaturas de la carrera ha resultado muy útil para el desarrollo del trabajo, así como el uso de herramientas de búsqueda de información.

7.2 Trabajos futuros

Una de las cosas que habría que añadir sería lo referente a las instrucciones *built-in* como una extensión aplicación. Debido a la manera que esta creada ahora mismo la aplicación, el uso de estas instrucciones genera excepciones ya que no es capaz de

reconocer este tipo de instrucciones correctamente y considera que son errores. La principal mejora de esta aplicación sería introducir estas funciones modificando las instrucciones del código rever referentes a este tipo de llamadas y del propio código Java para que permitan la integración .

Sería también bueno en un futuro alguna opción que permitiera la visualización de los resultados en una forma más gráfica, como por ejemplo como de un árbol de ejecución.

Bibliografía

Colmerauer,Roussel,1970: La naissance de Prolog; 1992

INTRODUCCIÓN AL PROLOG
http://informatica.uv.es/iiguia/LP/teoria/apuntes/cuatr2/prolog.pdf

Prolog https://es.wikipedia.org/wiki/Prolog#cite_note-1

3.Prolog https://www.uv.mx/personal/aguerra/files/2020/09/pia-03.pdf

Wielemaker,1987: SWI-Prolog Reference Manual https://www.swi-prolog.org/pldoc/doc_for?object=manual

Germán Vidal. Reversible Computations in Logic Programming. In Proc. of the 12th International Conference on Reversible Computation (RC 2020). Springer LNCS 12227, pp. 246-254, 2020.

Programing in XPCE/Prolog
https://www.swiprolog.org/download/xpce/doc/prolog/userguide.pdf

Sanchez Campello: Programando la interfaz gráfica con XPCE/Prolog
http://www.dccia.ua.es/logica/prolog/docs/ProgGUI.pdf

Wielemaker; Anjewierden,1987: XPCE/PROLOG Course Notes https://www.swi-prolog.org/download/xpce/doc/coursenotes/coursenotes.pdf

Dushin, Singleton, Wielemaker;2004: JPL: A bidirectional Prolog/Java interface
https://www.swi-prolog.org/packages/jpl/

Booch, Rumbaugh, Jacobson;1998: Rational Unified Process (RUP)
http://ima.udg.edu/~sellares/einf-es2/present1011/metodopesadesrup.pdf

Identidad visual corporativa https://www.upv.es/perfiles/pas-pdi/identidad-corporativa-upv-es.html

SWI-Prolog's features https://www.swi-prolog.org/features.html

JPL an API between SWI-Prolog and the Java Virtual Machine https://jpl7.org/

Java Development Kit https://www.ibm.com/docs/es/i/7.3?topic=platform-java-development-kit