UNIVERSITAT POLITÈCNICA DE VALÈNCIA

etsinf

Escola Tècnica Superior d'Enginyeria Informàtica

Escola Tècnica Superior d'Enginyeria Informàtica
Universitat Politècnica de València

Diseño e implementación de una herramienta para determinar la cobertura de las pruebas en Prolog

TRABAJO FIN DE GRADO

Grado en Ingeniería Informática

Autor: Daniel Santamarina Puertas

Tutor: Germán Francisco Vidal Oriola

Curso 2022-2023

Resumen

Este Trabajo Fin de Grado aborda el diseño e implementación de una herramienta para medir la cobertura de las consultas realizadas en un programa Prolog. A diferencia de las herramienta existentes, este trabajo aporta como novedad la instrumentación del código para ofrecer una mayor portabilidad, medidas de cobertura mejoradas y una mayor facilidad de uso.

En el ámbito de Prolog, la secuencia de ejecución hace que sea un desafío determinar si las pruebas diseñadas para validar el comportamiento de un programa están cubriendo adecuadamente todo el código. Es esencial medir la cobertura en este contexto para entender cuánto del código está respaldado por casos de prueba y garantizar así la calidad del software desarrollado.

En la formulación del proyecto, se exploran diversas soluciones para el problema, profundizando en los posibles entornos, herramientas y en el diseño que la aplicación final podría tomar. La elección de una técnica de instrumentación se debe a que permite aprovechar las herramientas propias del lenguaje sin la necesidad de utilizar un entorno separado durante su uso.

El proceso de desarrollo adopta la metodología Scrum, adaptada específicamente para este problema. Esta metodología define un enfoque iterativo, en el que se realizan pequeños avances que facilitan la adaptabilidad ante los cambios y la nueva información adquirida durante el desarrollo.

En definitiva, este Trabajo Fin de Grado integra las herramientas del lenguaje Prolog en el proceso de instrumentación, proponiendo una alternativa mejorada a los métodos existentes de medición de cobertura de código. Esta herramienta destaca la capacidad de los lenguajes de programación lógica en la resolución de problemas complejos, y subraya la importancia de herramientas que fomenten la creación de software de calidad.

Palabras clave: testing, cobertura, pruebas, Prolog

Abstract

This undergraduate thesis addresses the design and implementation of a tool to measure the coverage of queries made in a Prolog program. Unlike existing tools, this work introduces the novelty of instrumenting the code to provide greater portability, improved coverage measurements, and ease of use.

Within Prolog, the sequence of execution makes it challenging to determine whether the tests designed to validate a program's behavior adequately cover all the code. It is essential to measure coverage in this context to understand how much of the code is backed by test cases, ensuring the quality of the developed software.

In the project's formulation, various solutions to the problem are explored, delving into potential environments, tools, and the design the final application might take. The choice of an instrumentation technique is made because it allows leveraging the language's inherent tools without the need to use a separate environment during its use.

The development process adopts the Scrum methodology, specifically adapted for this problem. This methodology defines an iterative approach, in which small advances are made that facilitate adaptability in the face of changes and new information acquired during development.

In conclusion, this undergraduate thesis integrates the tools of the Prolog language into the instrumentation process, proposing an enhanced alternative to existing code

coverage measurement methods. This tool highlights the capability of logic programming languages in solving complex problems and emphasizes the importance of tools that promote the creation of quality software.

Key words: testing, coverage, Prolog

Índice general

Índice de figuras

Índice de tablas

CAPÍTULO 1
Glosario de Prolog

Átomo Una constante simbólica sin valor numérico.

Variable Un símbolo que representa un valor no especificado y que comienza con una letra mayúscula o con un guion bajo.

Sentencia Una declaración en Prolog, que puede ser una cláusula (hecho o regla) o una consulta.

Hecho Una declaración que establece una sentencia que es cierta.

Regla Una sentencia que relaciona antecedentes con consecuentes utilizando ":-" para denotar "si".

Consulta Una pregunta que se plantea al sistema Prolog para verificar si una afirmación es cierta.

Unificación El proceso por el cual Prolog intenta hacer que dos términos sean iguales, sustituyendo las variables según sea necesario.

Instanciación Proceso mediante el cual se asigna un valor específico a una variable en Prolog. Una vez instanciada, la variable ya no puede ser unificada con un término que no sea idéntico al valor asignado, a menos que se realice backtracking.

Búsqueda de la solución Proceso mediante el cual Prolog intenta satisfacer una consulta al tratar de unificarla con las cláusulas (hechos y reglas) disponibles en una base de conocimientos. Esta búsqueda finaliza cuando Prolog encuentra una solución que satisface la consulta o determina que no existen soluciones posibles.

Backtracking El mecanismo por el cual Prolog retrocede para encontrar más soluciones o para recuperarse de caminos de búsqueda fallidos.

Punto de elección Un punto en el proceso de ejecución donde Prolog podría tomar diferentes caminos al buscar soluciones. Si un camino no lleva a una solución, Prolog realiza backtracking a su último punto de elección y prueba un camino diferente.

Predicado Una relación representada por su nombre y aridad (por ejemplo, contiene/2).

Clausula Uno de los hechos o reglas que forman parte de la definición de un predicado.

Aridad Número de argumentos de un predicado.

Base de conocimientos Una colección de hechos y reglas, se refiere a la base de conocimientos también como un programa Prolog.

Lista Una estructura de datos compuesta que contiene una serie de términos.

Operador Un símbolo que realiza una función específica, como la unificación o la comparación.

Corte Un operador especial en Prolog (representado por el símbolo "!") que controla el backtracking.

CAPÍTULO 2
Introducción

El lenguaje de programación es la herramienta fundamental del profesional de la informática, ya que permite traducir los problemas en órdenes ejecutables por un computador. A medida que estos lenguajes han evolucionado, han tratado de acercarse a la forma en la que la persona programadora expresa sus ideas, proporcionando abstracciones y permitiendo la creación de programas más complejos.

Los lenguajes lógicos [Jackson, 2012] buscan dar un paso más en esta dirección, permitiendo la expresión de un problema como una serie de declaraciones y proporcionando la capacidad de inferencia de nuevos conocimientos en base en estas.

Sin embargo, esta misma naturalidad también puede presentar dificultades al depurar el código y detectar resultados inesperados. Por ello, se debe contar con un entorno de pruebas robusto, que proporcione las herramientas necesarias para llevar a cabo un análisis holístico.

En este trabajo, se aborda el diseño e implementación de una herramienta para determinar la cobertura de las pruebas en el lenguaje Prolog. El objetivo es ofrecer a la persona programadora una herramienta que complemente los procesos de detección de errores en el código y pueda ayudar a mejorar la calidad de los programas. Esta herramienta permitirá visualizar qué partes del código fuente han sido ejecutadas en los casos de prueba, asegurándose así de que se alcanza la cobertura del código deseada.

2.1 Motivación

Aunque los lenguajes lógicos, como Prolog [Apt, 1997], buscan proporcionar abstracciones que facilitan la creación de programas más robustos y seguros, a menudo carecen de herramientas para realizar un análisis riguroso del código [Brna et al., 1991]. Este trabajo busca proporcionar una herramienta que complemente a las existentes y que pueda ayudar a la persona programadora a mejorar la calidad del software desarrollado en Prolog.

El paradigma lógico y el lenguaje de Prolog se enseñan principalmente en entornos académicos, siendo su utilización en entornos comerciales reducida. Como herramienta de pruebas, se busca destacar la capacidad del lenguaje para permitir el desarrollo de software riguroso y seguro.

Al contribuir a la comunidad de Prolog este trabajo busca también unirse a los proyectos existentes para impulsar el aprendizaje y la exploración del lenguaje.

Por otro lado, mi motivación personal para la realización de este trabajo ha sido la de explorar con más detalle un lenguaje nuevo y distinto a aquellos que he utilizado

habitualmente durante la carrera para desarrollar proyectos y aplicaciones. Durante la carrera he disfrutado mucho de las asignaturas de programación, pero he sentido que no llegaba a un conocimiento muy profundo en la materia más allá del requerido para utilizar los conceptos. Este trabajo ha sido una oportunidad para explorar un lenguaje de una manera más detenida, asegurándome de que entiendo cada paso en el proceso.

Otro de los motivos por los que elegí realizar este trabajo fue la posibilidad de hacer una aportación a la comunidad de Prolog. A nivel personal, pienso que los lenguajes lógicos tienen un potencial muy grande en la informática, especialmente en el ámbito de la representación del conocimiento, un área cada día más popular por su aplicación en el contexto de la IA. A pesar de tener una dificultad inicial de aprendizaje mayor, estos lenguajes permiten una expresión del código más fluida y coherente con la naturaleza del problema que resuelven.

2.2 Objetivos

El objetivo general de este trabajo es el desarrollo de una herramienta que ayude a la persona programadora de Prolog a realizar pruebas sobre su código, proporcionando información adicional que le permita detectar posibles errores y mejorar la calidad del software.

Los objetivos específicos consisten en el desarrollo de una herramienta que cumpla con los siguientes requisitos:

- Deberá mostrar la información de forma clara y directa, siguiendo las convenciones habituales de Prolog.
- Deberá proporcionar un método de ejecución rápido y simple, propio de un entorno de desarrollo.
- Deberá medir la ejecución de todas las cláusulas del programa, permitiendo la elección de la consulta a lanzar.
- Deberá aceptar el lanzamiento de múltiples consultas, permitiendo la visualización de la cobertura acumulada en cualquier punto.
- Deberá poder manejar un subconjunto amplio del lenguaje de Prolog, admitiendo operaciones extra-lógicas como "cut", "assertz" o "fail".
- Deberá disponer de la capacidad de realizar medidas de la cobertura de distintas características, como la unificación de cláusulas y la instanciación de argumentos.

2.3 Impacto esperado

Se espera que esta herramienta mejore la calidad y optimice el esfuerzo invertido en el desarrollo de pruebas para Prolog. Al ofrecer una alternativa más precisa y potente para probar y analizar el código, se podría potenciar la calidad del software, detectar y corregir un mayor número de errores, reducir el tiempo de desarrollo y mejorar las aplicaciones en Prolog.

Como se ha detallado en el apartado de motivación, se espera que la herramienta fortalezca la percepción de Prolog como un lenguaje de programación viable tanto desde una perspectiva comercial como académica. Su impacto se extiende más allá de simples

mejoras técnicas en el ámbito de Prolog, impulsando también un cambio en la percepción y adopción del lenguaje.

Muchas herramientas de cobertura de código son específicas para una implementación de Prolog. Esta herramienta busca ofrecer un método que sea compatible con los estándares del lenguaje y que sea fácilmente portable a otras implementaciones de Prolog. Así, se espera que, con mínimas modificaciones, esta herramienta pueda utilizarse en diversos entornos de Prolog y ofrecer su funcionalidad a una base de personas usuarias más amplia.

2.4 Metodología

Dado que este trabajo sigue el diseño y la implementación de una herramienta con usos prácticos, se seguirá una de las metodologías estudiadas en la rama de ingeniería del software. En este caso se elegirá una metodología ágil, ya que se busca una metodología flexible adaptada a proyectos donde el producto final no está completamente definido desde el inicio.

Concretamente, se utilizará la metodología ágil Scrum, adaptando los roles para un equipo de desarrollo formado por un solo desarrollador, ya que esta permitirá avanzar el proyecto en iteraciones cortas y enmarcar las reuniones con el tutor como reuniones Scrum.

Dentro de esta metodología se contempla un periodo inicial de unas 100 horas para el aprendizaje del entorno Prolog y el intérprete de SWI-Prolog elegido para el desarrollo.

Una vez comenzado el desarrollo, este se divide en iteraciones de unas dos semanas de duración llamadas *Sprint*, donde se definen tareas específicas relacionadas con los objetivos. Al finalizar cada *Sprint*, se realiza una reunión con el tutor para presentar el trabajo completado, revisar el proceso (*Sprint Review*) y proponer mejoras y cambios para el próximo *Sprint* (*Sprint Planning*).

El enfoque del desarrollo se centrará en tener un prototipo funcional al final de cada *Sprint*. Para definir estos prototipos se simplificarán los objetivos para adaptarlos a este proceso. En la Sección 4.4 se muestran los objetivos adaptados para las iteraciones, así como los resultados de cada *Sprint*.

2.5 Estructura de la memoria

A continuación se presentará la estructura del resto del documento, proporcionando una pequeña explicación de las secciones:

- Glosario de Prolog: Aquí se definen algunos de los términos utilizados en la memoria para describir partes del lenguaje de Prolog, con el objetivo de ayudar a la lectura.
- Estado del arte: Se examina el contexto tecnológico de la herramienta, describiendo el entorno en el que se emplea y documentando otras herramientas con características y propósitos similares.
- Análisis del problema: Se detalla el problema a resolver, las necesidades asociadas, los objetivos y la metodología. Se identifican las soluciones posibles y se especifica la opción elegida.

- Diseño de la solución: Se expone cómo abordar el problema, detallando la estructura adoptada, los componentes principales y las herramientas y tecnologías seleccionadas.
- Desarrollo de la solución propuesta: Se documenta el proceso de desarrollo e implementación de la solución, resaltando las variaciones con respecto a la metodología inicialmente propuesta. Se muestran los resultados, tanto parciales como finales, así como los elementos de diseño relevantes para la solución.
- Conclusiones: Se presentan las conclusiones derivadas del trabajo, evaluando si se cumplieron los objetivos y reflexionando sobre el trabajo efectuado. Se expone también la relación del proyecto con los estudios realizados y se proponen líneas de investigación para futuros trabajos.

CAPÍTULO 3
Estado del arte

3.1 Presentación de Prolog

Prolog es un lenguaje de programación declarativo basado en la lógica de predicados. En Prolog, los programas se definen mediante hechos y reglas que establecen relaciones y conceptos, y se utiliza un mecanismo basado en unificación para inferir conclusiones a partir de ellos.

Los programas en Prolog se componen de definiciones de predicados. Por ejemplo `contiene/2` será un predicado de nombre *contiene* con 2 argumentos.

Si utilizamos este predicado para indicar qué ingredientes forman parte de una receta, podemos definir, por ejemplo, estas 3 cláusulas:

```
contiene(paella, arroz).
contiene(paella, conejo).
contiene(hervido, patata).
```

Es decir, en este programa `contiene(X, Y)` es cierto solo cuando `{X: paella, Y: arroz}`, `{X: paella, Y: conejo}` o `{X: hervido, Y: patata}`, indicando que el plato `X` contiene el ingrediente `Y`. Estas cláusulas, que muestran una afirmación, se denominan *hechos*.

Las *reglas* en Prolog establecen relaciones condicionales. Una regla se compone de una cabeza, que establece un hecho, y un cuerpo, que establece las condiciones para que el hecho sea verdadero. Por ejemplo, se puede definir el predicado `ingrediente/1`, que será verdadero si su argumento es un ingrediente. Esto se puede conseguir con la siguiente regla:

```
ingrediente(X) :- contiene(_,X).
```

Esta regla establece que `X` es un ingrediente si existe un hecho `contiene/2` cuyo segundo argumento es igual a `X`.

Para ejecutar un programa Prolog, se pueden hacer consultas al intérprete para determinar si un hecho es verdadero o falso. Por ejemplo, para probar si `patata` es un ingrediente, se puede hacer la siguiente consulta:

```
?- ingrediente(patata).
```

Prolog tratará de demostrar que hay una cláusula del predicado `contiene/2` donde el segundo argumento es `patata`. Al ser la tercera cláusula de `contiene/2` un hecho cuyo segundo argumento es `patata`, responderá `verdadero`.

En la búsqueda de soluciones, Prolog puede crear puntos de decisión cuando existen varias cláusulas que unifican con una llamada. Así, se puede explorar el árbol de posibles soluciones con una búsqueda "primero en profundidad" (*depth-first*) y volver atrás si una elección no resulta posible. Por ejemplo, añadiendo los predicados:

```
carne(conejo).

receta_con_carne(X) :- contiene(X,Y), carne(Y).
```

La interpretación de `receta_con_carne/1` es la siguiente: `X` es una receta con carne si existe una cláusula de `contiene/2` cuyo primer argumento es `X` y existe una cláusula de `carne/1` con su argumento igual al segundo de `contiene/2`. Ante la consulta:

```
?- receta_con_carne(paella).
```

Prolog buscará una cláusula de `contiene/2` para la que el primer argumento es `paella`, al encontrar dos soluciones posibles `{X: paella, Y: arroz}` y `{X: paella, Y: conejo}` seleccionará la primera, buscando demostrar el hecho `carne(arroz)`. Como este hecho no existe, pero había una alternativa pendiente más arriba en el árbol, Prolog hará *backtracking* y probará la segunda ruta. En este caso, el hecho `carne(conejo)` es cierto y la consulta verdadera, por lo que Prolog devolverá verdadero.

3.2 Depuración en SWI-Prolog

SWI-Prolog[1] es uno de los sistemas de Prolog más populares utilizado tanto en la industria como en la academia [Wielemaker et al., 2012]. Se puede ejecutar en varias plataformas e incluye una variedad de bibliotecas y herramientas.

SWI-Prolog utiliza el predicado `trace/0` para activar el depurador. Este depurador ofrece una serie de herramientas básicas para analizar la traza de una ejecución, facilitando la identificación y corrección de errores en el código. Aunque la herramienta proporciona una visión minuciosa de la ejecución, no da información sobre la cobertura general del código. Para complementar esta limitación, SWI-Prolog ofrece la biblioteca `test_cover`, que permite visualizar la cobertura general para una consulta dada.

La biblioteca `test_cover`, formada por el predicado `show_coverage/2`, muestra el número de ejecuciones de cada cláusula de un programa tras realizar una consulta. Distinguiendo entre cláusulas que han finalizado con éxito y aquellas que han fracasado, proporciona una visión general de la cobertura que facilita la identificación de las áreas que requieren cambios o mejoras.

Por ejemplo, ante la consulta planteada anteriormente:

```
?- receta_con_carne(paella).
```

`show_coverage/2` creará el siguiente archivo:

```
++1  contiene(paella, arroz).
###  contiene(hervido, patata).
++1  contiene(paella, conejo).

###  ingrediente(X) :- contiene(_,X).
```

[1] `https://www.swi-prolog.org/`

```
++1  carne(conejo).

++1 receta_con_carne(X) :- +1*2 contiene(X,Y), +1-1 carne(Y).
```

Las anotaciones añadidas por el predicado tienen los siguientes significados:

```
###     Cláusula nunca ejecutada
++N     Cláusula seleccionada N veces y siempre exitosa
--N     Cláusula seleccionada N veces y nunca exitosa
+N-M    Cláusula exitosa N veces y fallida M veces
+N*M    Cláusula seleccionada N veces y exitosa M veces
```

La principal motivación en la creación de la biblioteca `test_cover` y el predicado `show_coverage/2` es facilitar el desarrollo y gestión de los casos de prueba, también conocidos como *Unit Tests*. Durante el proceso de desarrollo, es importante garantizar que las consultas de prueba propuestas para evaluar el código tengan una cobertura completa de los predicados definidos. De lo contrario, los predicados sin cobertura podrían contener errores que pasarían inadvertidos.

3.3 Depuración en otros lenguajes lógicos

Logtalk

Logtalk [Moura, 2009] es un lenguaje de programación orientado a objetos que extiende Prolog y cuyo código es ejecutable en SWI-Prolog. Logtalk ofrece una herramienta *coverage* que, a través del predicado `logtalk_coverage/0`, muestra información de la cobertura de una consulta. Esta herramienta ofrece más información que `show_coverage/2`, por ejemplo, permite diferenciar entre las cláusulas que han unificado con una consulta y aquellas que han sido seleccionadas.

ECLiPSe

ECLiPSe [Schimpf and Shen, 2012] es un lenguaje de programación lógica con restricciones basado en Prolog y diseñado para resolver problemas de satisfacción de restricciones y optimización. Como Logtalk y SWI-Prolog, ECLiPSe tiene una herramienta de cobertura de código llamada *coverage_analyzer* para proporcionar información de cobertura. Además de la funcionalidad de `show_coverage/2`, permite obtener información sobre las restricciones que guían la unificación y los modos que toman los argumentos en el proceso de la búsqueda.

3.4 Crítica al estado del arte

A pesar de ser una solución conocida y disponible para la programación en Prolog, la herramienta `test_cover` de SWI-Prolog ofrece una serie de limitaciones funcionales que dificultan un análisis exhaustivo y preciso de la cobertura de las pruebas.

Una de las restricciones es que `test_cover` mide la cobertura después de que SWI-Prolog haya unificado con un predicado. Esta limitación impide medir qué cláusulas unificarían en una consulta, tal y como hace el predicado `logtalk_coverage/0` de Logtalk.

Esto puede ser de especial importancia para garantizar que los *Unit Tests* no solo cubran las cláusulsa de un predicado, sino también la unificación con distintas combinaciones de predicados.

Además, la implementación de la herramienta presenta problemas adicionales, ya que no está bien documentada y no permite realizar cambios y mejoras a su funcionamiento fácilmente. Por otro lado, su uso incurre en un tiempo de ejecución significativamente más lento (hasta 10 veces), no permite el uso del depurador durante su ejecución y no tiene soporte para hilos.

La implementación hace uso de *tracer hooks*, una función interna a SWI-Prolog y no contenida en el estándar ISO [2] del lenguaje, para las medidas de cobertura. Esto impone grandes limitaciones sobre la portabilidad de la herramienta, ya que no es posible ejecutarla en otros sistemas de Prolog distintos a SWI-Prolog.

3.5 Propuesta

Este trabajo propone el diseño y desarrollo de una herramienta alternativa a `test_cover` que mejore sus capacidades y aborde sus carencias. En la documentación de SWI-Prolog se contempla una alternativa a la herramienta a través del uso de un meta-intérprete. Este trabajo explora esta opción, así como la instrumentación del código, valorando la eficiencia, claridad y versatilidad de cada enfoque.

La herramienta propuesta aspira a medir la cobertura de la ejecución sin depender de funciones específicas de la implementación SWI-Prolog. Además, se buscará ampliar sus funciones, proporcionando información más detallada, como los ”modos” de los argumentos y las cláusulas unificadas en el proceso de búsqueda de soluciones. Estos cambios facilitarán la identificación de posibles errores y permitirán una comprensión más completa de la cobertura de los *Unit Tests*.

[2] `https://www.iso.org/standard/21413.html`

CAPÍTULO 4
Análisis del problema

4.1 Especificación de requisitos

En cualquier desarrollo de software, es fundamental llevar a cabo un análisis completo del problema desde la perspectiva de todas las personas involucradas. En esta Sección, se detalla dicho análisis para la herramienta propuesta que mide la cobertura de código de los casos de prueba en Prolog. La estructura de esta especificación sigue el estándar IEEE 830-1998 [IEEE, 1998].

4.1.1. Propósito

En esta Sección se busca medir los requisitos funcionales y no funcionales del sistema con el fin de obtener una descripción precisa, coherente y correcta del sistema que se desarrollará. Dichos requisitos estarán expresados de manera que puedan ser priorizados y verificados, facilitando así la guía del proceso de desarrollo y vinculando los objetivos específicos con resultados tangibles.

4.1.2. Alcance

La herramienta propuesta tiene como objetivo informar sobre la cobertura de la ejecución en Prolog, permitiendo analizar este aspecto en la ejecución de diversos programas para asistir en el proceso de desarrollo. Esta herramienta está diseñada para apoyar los procesos detallados a continuación:

- Visualizar la ejecución de uno o varios predicados.
- Eliminar cláusulas y predicados innecesarios en un programa Prolog.
- Diseñar casos de prueba que aseguren un mayor grado de cobertura.

A través de los procesos mencionados, es evidente que el propósito de la herramienta es ofrecer apoyo en el ámbito del desarrollo de software. Por esta razón, su diseño debe ser el de una herramienta no intrusiva, capaz de integrarse en sistemas más avanzados y operar en diferentes entornos. Esto se debe a que coexistirá con otras herramientas que facilitan el proceso de desarrollo.

Dicha característica es crucial al analizar los casos de uso, ya que comprender y analizar los entornos en los que se lleva a cabo el desarrollo de Prolog facilitará el establecimiento de una relación óptima entre la herramienta propuesta y las ya existentes. Dicho análisis se encuentra en el apartado 4.1.3, donde se examinan las preferencias de

las personas usuarias en comparación con las herramientas más populares en el ámbito de Prolog.

4.1.3. Características de uso

En esta Sección se identifican y delimitan algunas características que las personas usuarias poseen. Para adaptar los casos de uso a las necesidades de estas personas, es esencial analizar los conocimientos técnicos y preferencias de formato que puedan tener.

La persona usuaria deberá contar con un conocimiento intermedio de programación, específicamente en el lenguaje Prolog. Deberá entender la traza de ejecución de una consulta en Prolog y conocer términos como ”predicados”, ”cláusulas”, ”unificación”, ”instanciado”, entre otros.

Los entornos como SWI-Prolog y otras implementaciones populares de Prolog, se ejecutan principalmente a través de una interfaz de consola. Por ende, se deduce que si la persona usuaria tiene conocimientos de Prolog, también poseerá habilidades para trabajar en un entorno de línea de comando. Cambiar una herramienta desde una interfaz de comandos a una interfaz gráfica podría generar ineficiencia y confusión en su uso, por lo que se supone que preferirán una herramienta que permanezca en dicho entorno.

Además de los conocimientos técnicos necesarios para el uso de la herramienta, se asume que la persona usuaria está familiarizada con otras herramientas de monitorización en ejecución, como la función `trace` presente en el estándar de Prolog. De manera similar a la interfaz de consola, estas herramientas son esenciales para el uso de Prolog y, por tanto, se consideran disponibles para la persona usuaria en la comprensión de los resultados de la cobertura.

4.1.4. Casos de uso

Los casos de uso seguirán la estructura de los procesos detallados en el apartado ”Alcance” (4.1.2). Aunque estos procesos se desarrollen de manera continua durante la creación de software, su secuencia concuerda con el orden en el que suelen presentarse en el desarrollo.

Durante la fase inicial de desarrollo, el proceso de visualizar la ejecución de determinados predicados ayuda a comprender mejor cómo funciona el código y proporciona un soporte complementario a la visualización de trazas ya existente para mostrar detalles del código.

En etapas más avanzadas del desarrollo, cuando los predicados involucrados en una ejecución se multiplican, pueden surgir segmentos del código que se vuelven obsoletos, ocultados por una condición o una llamada modificada. En este contexto, la traza de la ejecución solo mostrará las secciones del código que se ejecuten y no examinará aquellas que no se invoquen. Esta situación, sumada a la existencia de programas más extensos donde la traza no proporciona una vista completa de la ejecución sino solo de un fragmento, resalta la relevancia de la segunda función de la herramienta en esta fase.

Dentro de los patrones convencionales de desarrollo, es usual establecer una serie de casos de prueba en las etapas finales del desarrollo [Mills, 1976]. Esto garantiza que el código en uso, así como sus futuras modificaciones, no alteren los resultados esperados en las distintas secciones del sistema. Por ello, es esencial asegurarse de que los casos de prueba cubran todo el código. Si un predicado o una cláusula no se unifica durante la ejecución de las pruebas, cualquier cambio en su contenido no influiría en los resultados de las mismas.

A continuación, se describen los casos de uso para cada una de estas funciones.

Nombre	CDU01
Actuación	Desarrollo
Descripción	Proceso para visualizar la ejecución de un predicado Prolog
Precondiciones	Tener un fichero de código ".pl" que se desee medir
Postcondiciones	Se conoce la cobertura de las distintas cláusulas, identificando aquellas que no han sido ejecutadas y el número de ejecuciones de las que sí lo han sido. Además, se proporciona información adicional sobre la cobertura, relacionada con la unificación entre la consulta y las cláusulas.
Flujo Normal	1. Se accede al entorno de medición de cobertura mediante la interfaz de línea de comandos. Durante este acceso, se especifican los parámetros de ejecución, la dirección del fichero a analizar y el predicado que se quiere medir. 2. El sistema presenta en la línea de comandos una tabla que muestra las cláusulas del programa y el número de ejecuciones que ha tenido cada una. Esta información se subdivide y se acompaña de valores complementarios para facilitar su comprensión. 3. El sistema muestra el resultado de la ejecución: *true* si ha sido exitosa y *false* en caso contrario. 4. El proceso finaliza.
Flujo Alternativo	Si el fichero proporcionado es erróneo o no existe: 2. El sistema muestra un mensaje de error. 3. El proceso acaba. Si el fichero proporcionado contiene un error de ejecución: 2. El sistema muestra el error de ejecución que ha causado la terminación forzosa. 3. El proceso acaba.

Tabla 4.1: Caso de uso CDU01

Nombre	CDU02
Actuación	Desarrollo
Descripción	Proceso para eliminar cláusulas y predicados innecesarios en un programa Prolog
Precondiciones	1. Tener un fichero de código ".pl" que contenga el código Prolog del sistema a medir
Postcondiciones	La cobertura de los predicados y cláusulas definidos en un fichero ".pl" ha sido determinada. Se identifican los predicados que no se han ejecutado y se realiza un análisis individual de la cobertura de cada predicado y cláusula.
Flujo Normal	1. Se accede al entorno de medición de cobertura a través de la interfaz de la línea de comandos. Durante este acceso, se especifican los parámetros de ejecución, la dirección del fichero a analizar y la consulta que se desea aplicar al código. 2. El sistema presenta en la línea de comandos una tabla con las cláusulas del programa y el número de veces que se han ejecutado cada una. Esta información se encuentra organizada y acompañada de datos complementarios para facilitar su comprensión. 3. El sistema muestra el resultado de la ejecución: *true* si la ejecución ha sido exitosa o *false* si no lo ha sido. 4. El proceso finaliza.
Flujo Alternativo	Si el fichero no existe o la ejecución lanza un error: 2. El sistema muestra un mensaje de error describiendo la causa. 3. El proceso acaba.

Tabla 4.2: Caso de uso CDU02

Nombre	CDU03
Actuación	Pruebas
Descripción	Proceso para medir la cobertura de los casos de prueba sobre un sistema Prolog
Precondiciones	1. Tener un fichero de código ".pl" que contenga el código Prolog del sistema a medir 2. Tener un fichero ".plt" que contenga una lista de consultas para el sistema, las cuales constituyen los casos de prueba.
Postcondiciones	Se conoce la cobertura de los casos de prueba presentes en el fichero ".plt" respecto al sistema. Se identifican los predicados que no se han ejecutado y se proporciona un análisis detallado de la cobertura de cada predicado y cláusula.
Flujo Normal	1. Se accede al entorno de medición de cobertura mediante la interfaz de la línea de comandos. Durante este acceso, se especifican los parámetros de búsqueda, así como la dirección del fichero a analizar y el fichero que contiene los casos de prueba en forma de una lista de consultas. 2. El sistema muestra en la línea de comandos una tabla con las cláusulas del programa y el número de veces que se han ejecutado. Esta información se presenta de manera subdividida y acompañada de datos complementarios para facilitar su comprensión. 3. El sistema muestra el resultado de la ejecución: *true* si la ejecución ha sido exitosa o *false* en caso contrario. 4. El proceso finaliza.
Flujo Alternativo	Si alguno de los ficheros no existe o la ejecución lanza un error: 2. El sistema muestra un mensaje de error detallando la causa. 3. El proceso finaliza.

Tabla 4.3: Caso de uso CDU03

El propósito de esta herramienta es servir como apoyo en el proceso de desarrollo, por lo que es esencial que su acceso sea rápido y sencillo. La herramienta de trazado integrada de SWI-Prolog se activa mediante un comando en la consola integrada. Una vez activada, cualquier consulta efectuada mostrará primero la traza del programa y, a continuación, los resultados de dicha consulta. La herramienta de medición de cobertura opera de forma similar: después de una llamada inicial que activa la medición, cualquier consulta subsiguiente, contenida en un predicado para activar la medición, primero mostrará una tabla con las medidas de cobertura y, luego, el resultado de la ejecución. Es decir, los tres casos de uso mencionados anteriormente (CDU01, CDU02, CDU03) se pueden modificar, reemplazando el acceso por línea de comando con uno integrado en la consola de Prolog.

El caso CDU04 ejemplifica esta modificación, aplicada a la visualización de la ejecución de uno o más predicados.

Nombre	CDU04
Actuación	Desarrollo
Descripción	Proceso para visualizar la ejecución de un predicado Prolog
Precondiciones	Tener un fichero de código ".pl" que se desee medir
Postcondiciones	Se conoce la cobertura de las distintas cláusulas, identificando aquellas que no han sido ejecutadas, así como el número de ejecuciones de las que sí lo han sido. Además, se proporciona información adicional sobre la cobertura, relacionada con la unificación entre la consulta y las cláusulas.
Flujo Normal	1. Se activa la medición de cobertura mediante una consulta en la consola de Prolog, donde se especifican los parámetros de ejecución. 2. En la consola de Prolog, se introduce el predicado que se desea medir así como una indicación para realizar la cobertura. 3. El sistema presenta en la línea de comandos una tabla que detalla las cláusulas del programa y el número de ejecuciones de cada una. Esta información se subdivide y se acompaña de valores complementarios para facilitar su comprensión. 4. El sistema muestra el resultado de la ejecución: *true* si la ejecución ha sido exitosa y *false* en caso contrario. 5. El proceso finaliza.
Flujo Alternativo	Si el predicatdo proporcionado contiene un error de ejecución: 2. La consola muestra el error de ejecución que ha causado la terminación forzosa.

Tabla 4.4: Caso de uso CDU04

4.1.5. Restricciones y dependencias

Como se estableció en apartados anteriores, el marco de la herramienta en entornos de desarrollo tiene como enfoque principal proporcionar información valiosa de manera rápida y eficiente. Las restricciones refuerzan este aspecto, facilitando una estructura de trabajo fluida entre los sistemas de Prolog comúnmente utilizados y este sistema en particular.

- El sistema debe funcionar de manera similar a los sistemas existentes en el entorno de Prolog.
- El sistema debe ser compatible con distintas versiones de Prolog sin requerir modificaciones a su funcionamiento.

- El sistema debe ofrecer información de tal manera que su ejecución sea fácilmente entendible.
- El sistema debe poder ejecutarse desde la consola de Prolog y desde la línea de comandos.
- La implementación del sistema debe seguir una estructura clara y modificable, permitiendo la realización de cambios y añadidos durante su uso.

Dada la naturaleza técnica de la aplicación, se han realizado una serie de suposiciones en su desarrollo. A continuación se detallan estas suposiciones:

- La persona usuaria tiene experiencia en el lenguaje Prolog, entiende sus estructuras básicas y sabe cómo funciona su ejecución.
- El código ya ha sido analizado desde diferentes enfoques, como trazas de ejecución y análisis de componentes, antes del análisis de cobertura.

En SWI-Prolog, es esencial el uso de la consola para hacer consultas básicas y obtener resultados de los programas. Además, para emplear estructuras básicas de Prolog, como el operador de corte '!', es imprescindible comprender cómo Prolog lleva a cabo la unificación interna de una consulta. En otras palabras, las suposiciones establecidas para el uso del sistema coinciden con el nivel técnico que se esperaría que tuviera.

La aplicación se desarrolla en el lenguaje Prolog utilizando solamente las funciones básicas del lenguaje, según lo definido en el estándar ISO/IEC 13211; por lo tanto, no se tiene ninguna dependencia.

4.1.6. Requisitos

Requisitos de interfaz

Interfaces de usuario: La interfaz debe ser compacta y directa, alineada con el estilo de otras herramientas de Prolog, como el "trace" de SWI-Prolog. Dada la naturaleza de la información a mostrar, es ideal presentar los datos de cobertura en forma de tabla.

Interfaces de software: El sistema en el que se ejecuta la aplicación debe contar con una versión actualizada de SWI-Prolog. Además, es necesario tener acceso a una interfaz de línea de comando para ejecutar SWI-Prolog y la herramienta de cobertura.

Requisitos funcionales

Medida de cobertura: Los predicados que se deseen medir para la cobertura deben estar contenidos en un único archivo. Si estos hacen referencia a otros archivos, la ejecución de los predicados importados se ignorará en la medición de cobertura.

Gestionar operaciones externas: Las llamadas a las librerías internas de SWI-Prolog o a predicados externos a los que se les está midiendo la cobertura, funcionarán sin alteraciones.

Medida de cobertura en la unificación: Se deben medir tanto la cobertura de las combinaciones de cláusulas relacionadas con un predicado, como la instanciación de los argumentos de los predicados previa a la unificación.

Resultado de consulta: La cobertura obtenida después de la ejecución debe mostrarse de forma visual y de fácil interpretación.

Medidas a predicados y consultas dinámicas: Las consultas a predicados realizadas de manera dinámica, como por ejemplo a través del predicado `call/1`, deben procesarse adecuadamente, considerando su cobertura si el predicado consultado está en el archivo. Los predicados declarados de forma dinámica no se tomarán en cuenta en la medida de cobertura.

Requisitos no funcionales

Rendimiento: La medición de la cobertura debe ser eficiente para ser adecuada en un entorno de pruebas.

Portabilidad: La herramienta debe ser compatible en cualquier plataforma o sistema donde SWI-Prolog pueda funcionar.

Fiabilidad: La aplicación debe ser resistente frente a predicados y estructuras que no pueda procesar, manteniéndolos en el programa y midiendo la cobertura solo sobre aquellos que pueda procesar.

Mantenibilidad: La herramienta debe poseer un diseño claro y documentación detallada que facilite futuras modificaciones. Además, debe incluir un conjunto de pruebas que cubran todas sus funcionalidades.

4.2 Identificación y análisis de soluciones posibles

Como se ha introducido en el apartado 2.4, este proyecto adopta la metodología Scrum para su desarrollo. Se trata de una metodología ágil e iterativa, en la que se persigue elaborar prototipos partiendo de requisitos básicos, para luego iterar y añadir más requisitos hasta alcanzar una solución completa.

Dada la complejidad del problema tratado en este trabajo, existen múltiples soluciones que lo abordan desde diferentes enfoques. A lo largo del desarrollo, no solo se analizaron las soluciones potenciales, sino que también se crearon prototipos funcionales para implementarlas. Aunque el desarrollo de estos prototipos siguió la metodología mencionada, la razón subyacente para su creación radica en el contexto académico del estudio y en la posibilidad de comprender el problema desde diversas perspectivas. En un contexto empresarial, este tipo de exploración se llevaría a cabo de forma más eficiente, descartando ciertas soluciones tras evaluar sus limitaciones y ventajas.

En esta Sección, se examinan las diferentes soluciones identificadas para el problema, presentando tanto las ventajas de su implementación como sus desventajas. El desarrollo de prototipos para cada una de estas soluciones facilitará su análisis desde una perspectiva de desarrollo de software, considerando las especificidades de cada implementación.

Para determinar la cobertura de código durante una ejecución, es indispensable recopilar datos sobre dicha ejecución. Esto implica que el programa debe reconocer qué predicados exploraría un intérprete Prolog al ejecutar una consulta. Asumiendo que ya existen intérpretes de Prolog, la divergencia entre las soluciones se centra en cómo se realiza esta ejecución. Por un lado, se puede optar por prescindir de los intérpretes actuales y diseñar uno nuevo que registre datos sobre la cobertura y los presente tras una consulta. Este intérprete podría desarrollarse en Prolog, aprovechando sus herramientas de unificación e interpretación para facilitar la creación de un nuevo intérprete en el mismo

lenguaje. Otra alternativa consiste en modificar los predicados del código Prolog que se quiere evaluar, agregando argumentos y predicados que almacenen la información requerida para determinar la cobertura. Una última opción sería adaptar un intérprete ya existente, como SWI-Prolog, empleando su estructura interna para añadir la funcionalidad que permita medir la cobertura.

4.2.1. Intérprete nuevo

La manera más directa de acceder a la traza de ejecución de una consulta en Prolog y medir la cobertura del código, es mediante un intérprete personalizado que recopile estos valores. Esta técnica ofrece la ventaja de ser totalmente independiente de cualquier intérprete ya existente. El intérprete diseñado podría ser usado en cualquier sistema que sea capaz de ejecutar el código. Asimismo, al tener un control total sobre su ejecución, facilita la medición, ya que las estructuras de ejecución se encuentran abstraídas del sistema Prolog.

No obstante, pese a estas ventajas, crear un intérprete para un lenguaje de programación es un desafío, ya que requiere el diseño de sistemas no triviales. La dificultad inherente a este proceso conlleva a que el intérprete pueda tener limitaciones en cuanto a solidez, mantenimiento y velocidad en comparación con intérpretes consolidados como el de SWI-Prolog. Por otro lado, los intérpretes populares ofrecen funciones de Prolog que no están presentes en las especificaciones del estándar ISO/IEC 13211, como la librería "test_cover" mencionada en la Sección 3.2. Al operar de manera independiente del entorno en el que estas herramientas están disponibles, la herramienta de cálculo de cobertura no podría ser aplicada a códigos que las integren. Esta perspectiva, al ser independiente de SWI-Prolog y otros intérpretes, también restringiría su facilidad de uso, pues no se ejecutaría dentro del entorno en el cual se desarrolla el código y que ya está disponible.

A pesar de las desventajas notables de esta implementación frente a otras alternativas, en el desarrollo de este estudio se ha construido un prototipo básico. En un entorno empresarial, siguiendo la metodología *Scrum*, este enfoque sería inapropiado, dado que la complejidad de la tarea es elevada y hay soluciones más óptimas disponibles. No obstante, desde una perspectiva académica, esta implementación ha sido de gran valor para el trabajo, puesto que exige una comprensión profunda del funcionamiento interno de Prolog. La experiencia adquirida a través de su desarrollo ha sido esencial para la solución final, tanto en su construcción como en su conceptualización.

El programa se ha desarrollado en el lenguaje Python y consta de aproximadamente 500 líneas de código. El sistema se divide en dos partes: en primer lugar, se procesa un programa Prolog que está escrito en un fichero de texto con la extensión ".pl", y posteriormente se realiza una consulta a este programa ya procesado.

La Figura 4.1 muestra el procesamiento del programa mediante uno de los ejemplos que se detalla en la Sección 3.1. A la izquierda, se muestra el flujo de ejecución y a la derecha, el resultado de cada tarea aplicado al ejemplo mencionado. En este flujo, primero se convierte el texto en una serie de tokens clasificados como átomos o símbolos. Luego, se efectúa una limpieza de la sintaxis corrigiendo estructuras especiales, como listas y el símbolo "[|]", para garantizar un formato uniforme. Tras obtener la lista de tokens filtrados, se determina la precedencia de los diferentes operadores, es decir, se aplican las reglas de ordenación establecidas por el estándar ISO/IEC 13211 (Sección 6.3.4.3) para homogeneizar el formato de los operadores a uno coherente de predicado. Finalmente, se utiliza la lista de tokens filtrados y reorganizados para instanciar los objetos correspondientes que representan cada predicado y átomo en el programa.

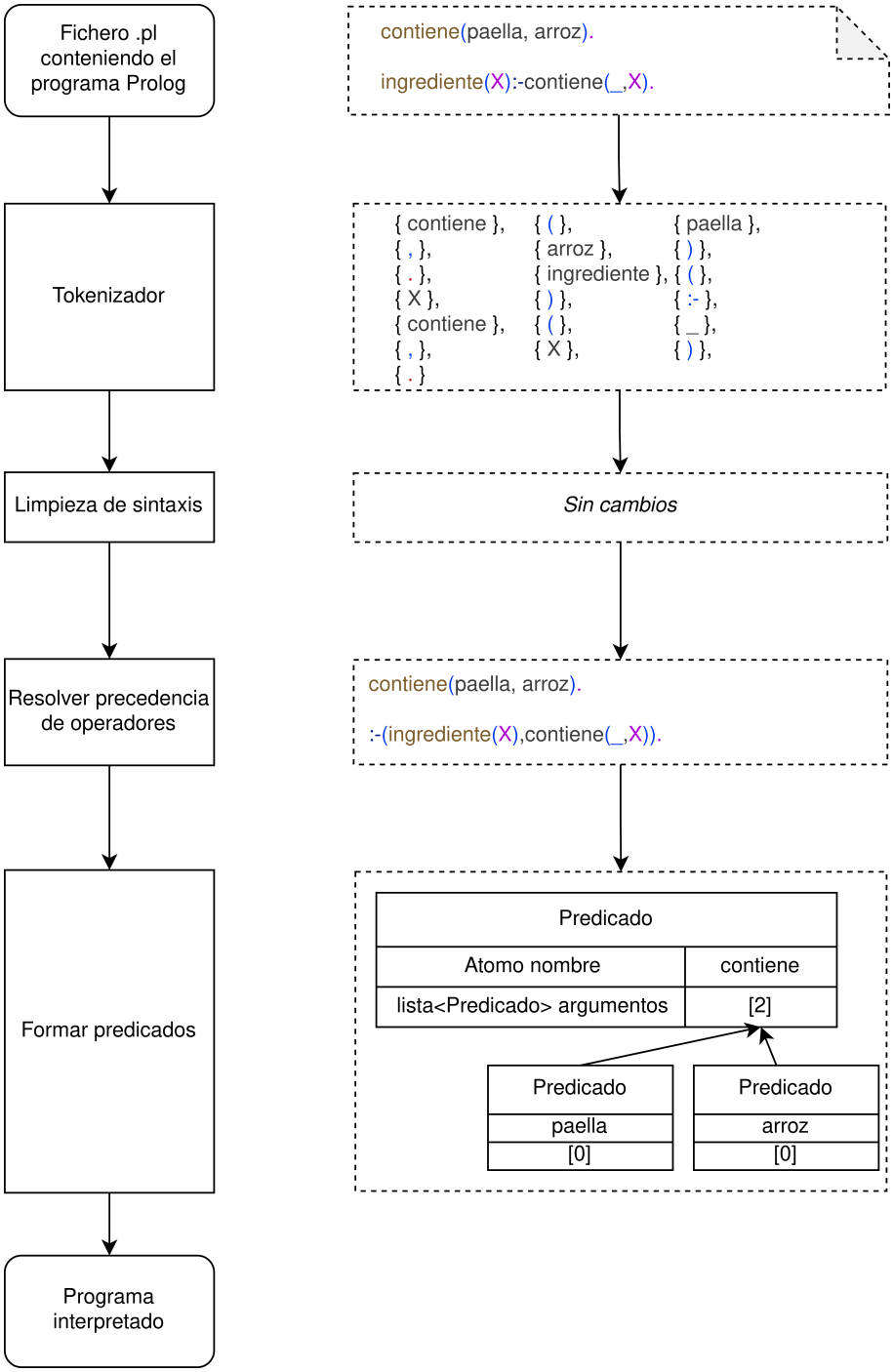

Figura 4.1: Flujo de la interpretación de un programa desde un fichero.

Tras el procesamiento, se obtiene una estructura de objetos en Python que representa el programa Prolog. Para ejecutar este programa Prolog, es decir, para realizar una consulta, es esencial implementar el procedimiento de unificación y búsqueda. La Figura 4.2 muestra el flujo de ejecución para resolver una consulta.

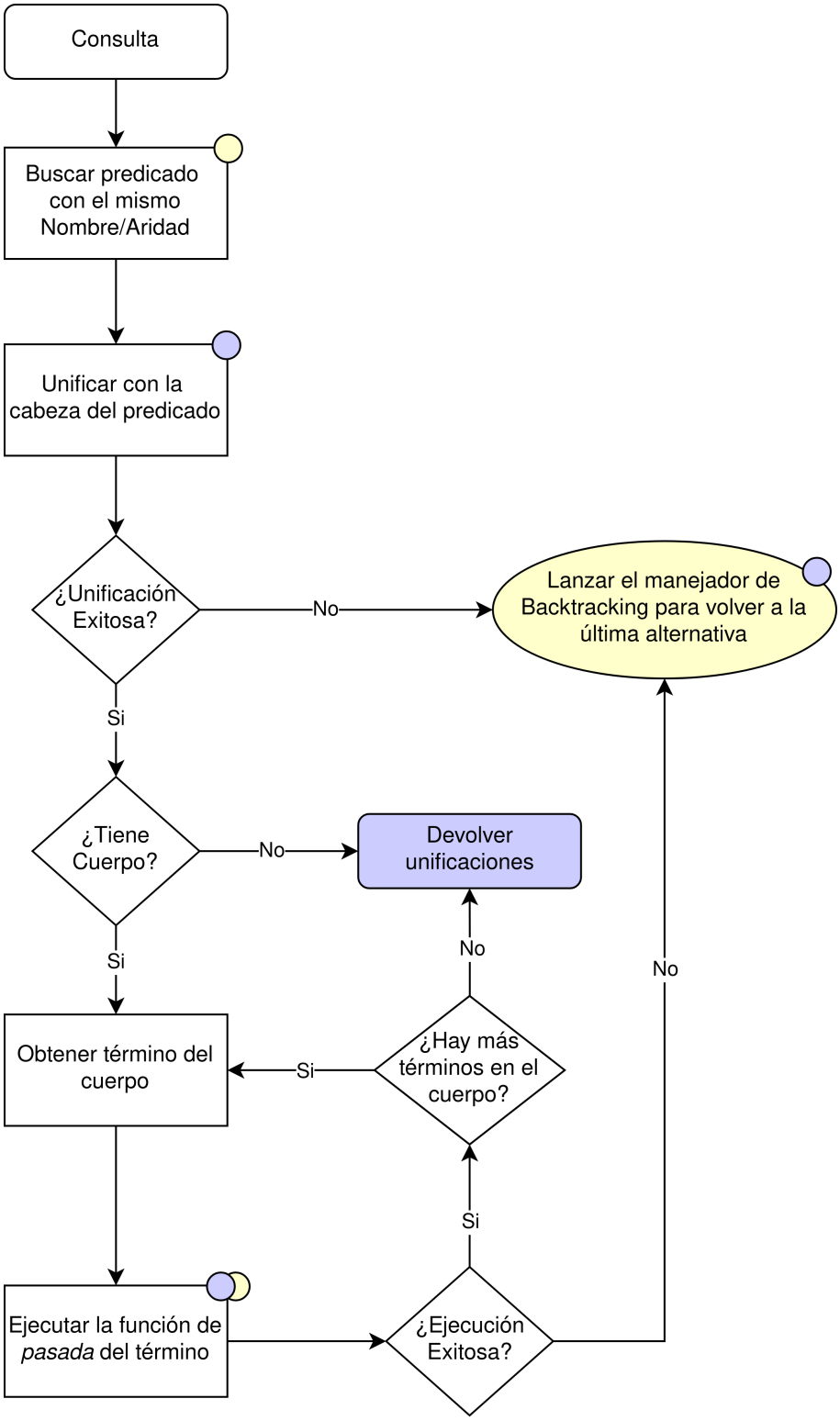

Figura 4.2: Flujo de ejecución para la resolución de una consulta.

Unificación

En este proceso, se parte de una consulta inicial, es decir, una llamada a un predicado Prolog. El sistema debe determinar la unificación de dicha llamada con las cláusulas del programa. Si no es posible unificarlo, el sistema debe retornar el átomo `false`. Este procedimiento es de naturaleza recursiva, ya que, para predicados definidos por reglas,

tras unificar con la cabeza, se necesita hacer otras consultas. Debido a esta característica, el proceso de registrar las unificaciones no es sencillo. Estas unificaciones deben ser independientes en distintos niveles de la recursión, pero conectadas dentro de una misma consulta. Por ejemplo, consideremos el siguiente programa::

```
foo(a, Y) :- bar(X, Y).
bar(c, b).
```

Si se realiza la consulta:

```
foo(X, Y).
```

esta unificará con `foo(a, Y)` con `{X: a}`. Al ser la cláusula de `foo/2` una regla, para verificar si esta es cierta, Prolog deberá realizar ahora la consulta `bar(X, Y)`, que unificará con `bar(c, b)` con `{X: c, Y: b}`. Así, las unificaciones finales para la consulta serán `{X: a, Y: b}`. El intérprete que se ha desarrollado sustituye las variables por pseudo-variables para distinguir entre las unificaciones de distintos niveles. Por ejemplo, las unificaciones del ejemplo después de la segunda consulta serían: `{X_1: a, Y_1: Y_2, X_2: c, Y_2: b}`.

En el intérprete, estas unificaciones se guardan en un objeto que puede ser modificado por los procesos durante la ejecución. En el flujo de ejecución de la Figura 4.2, estas modificaciones se muestran con el color azul. Es decir, los procesos marcados con un círculo azul pueden realizar alguna modificación sobre las unificaciones.

Búsqueda de solución

Una característica que aporta una complejidad adicional en el lenguaje Prolog es el *Backtracking* usado para buscar soluciones. Durante el proceso de unificación, el intérprete puede encontrar caminos alternativos de búsqueda. Por ejemplo, consideremos el siguiente programa:

```
foo(a) :- bar(a).
foo(b) :- bar(b).
bar(b).
```

y la consulta:

```
foo(X).
```

En este contexto, `foo(X)` puede unificar tanto con la primera como con la segunda regla del programa, ya que ambas son cláusulas del predicado `foo/1`. El intérprete optará por unificar con `foo(a) :- bar(a)`, dado que es la primera cláusula . Sin embargo, al existir una segunda cláusula de `foo/1` con la cual se puede unificar, el intérprete conservará un punto de elección, señalando la posibilidad de otras soluciones. Tras intentar la consulta `bar(a)`, no unifica con ninguna cláusula y, por tanto, falla. Entonces, el sistema consultará la lista de puntos de decisión y regresará a la última alternativa disponible. En este caso, la alternativa `foo(b) :- bar(b)` resulta exitosa, puesto que el hecho `bar(b)` se encuentra en el programa, dando lugar a la unificación `{X: b}`.

En el flujo de ejecución de la Figura 4.2, los procesos donde puede surgir un punto de elección están señalados con un círculo amarillo. Cuando se invoca el manejador de backtracking, este redirige la ejecución a uno de los procesos anteriores. Si no existe ningún punto de elección, la consulta retornará `false`, indicando que la solicitud no puede satisfacerse

Función de pasada

Un ejemplo fundamental para el funcionamiento de Prolog, mencionado en la Sección 3.1, es el operador *and*, que se representa en el cuerpo de una regla mediante el símbolo ",". Considerando el ejemplo:

```
foo :- bar, baz.
```

El cuerpo de la regla `foo/0` puede interpretarse como un único predicado al modificar la sintaxis del operador (este ajuste se realiza durante la lectura del programa):

```
foo :- ','(bar, baz).
```

El predicado `','` debe tenerse en cuenta durante la ejecución. Sin embargo, en este contexto, no se debe realizar una consulta del hecho `','(bar, baz)`. En su lugar, se consultan `bar` y `baz` de forma independiente. Para integrar estos comportamientos de forma modular en el proceso de ejecución, se invoca una función llamada *pasada*, que engloba todos los átomos del programa. A continuación, presentamos un fragmento de pseudocódigo para ilustrar la función de *pasada* de `','`:

```
def pasada(self, unificaciones):
    unificaciones_arg1 = self.arg1.pasada(unificaciones)

    unificaciones_arg2 = self.arg2.pasada(unificaciones_arg1)

    return unificaciones_arg2
```

Cobertura

En esta implementación, calcular la cobertura es sencillo: basta con guardar los predicados que acceden a la función *pasada*. Además, al tener acceso al funcionamiento interno de Prolog, es posible efectuar medidas más complejas, como la cobertura de los operadores que integran el sistema Prolog.

4.2.2. Intérprete dentro de Prolog

Programar un intérprete que facilite la cobertura dentro del lenguaje Prolog ofrece varias ventajas en comparación con una solución independiente. Dicho intérprete debe leer un fichero ".pl" que contenga el programa y traducir las líneas de código a un formato que sea interpretable. Al recibir una consulta, este intérprete debe llevar a cabo la unificación de la misma forma que lo haría el método tradicional, con el objetivo de encontrar una solución y evaluar la cobertura durante el proceso de búsqueda.

Aunque esta alternativa pueda parecer idéntica a la mencionada previamente, el flujo de ejecución y los pasos para su desarrollo son considerablemente distintos. Implementar el intérprete dentro del lenguaje Prolog permite acceder a las herramientas ya disponibles en el mismo para realizar la unificación, búsqueda y tokenización de predicados Prolog. El uso de estas herramientas facilita y agiliza el proceso de desarrollo. En la Figura 4.3, se ilustra el flujo adaptado de procesamiento del programa para este intérprete, aprovechando las herramientas que ofrece el lenguaje Prolog.

La función de lectura de predicados se ejecuta de manera similar al intérprete externo a Prolog. En ella, se definen los distintos tipos de estructuras y se simplifica la lista de

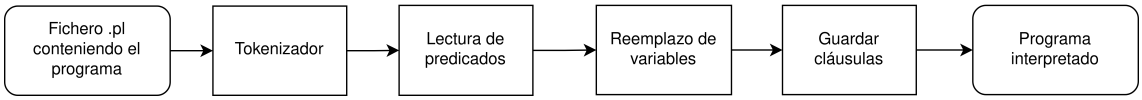

Figura 4.3: Flujo de procesamiento de un programa

términos que el tokenizador clasifica, para así generar una lista de cláusulas. Durante este proceso, se hace evidente una desventaja significativa de este método: si el intérprete se encuentra con una estructura desconocida, no podrá incluirla en la ejecución, por lo que no podrá procesarla correctamente.

La función de reemplazo de variables es necesaria en la ejecución, ya que las cláusulas que se procesan del fichero se almacenan como predicados dinámicos en la base de conocimientos utilizando el predicado `assertz/1`. Para posteriormente unificar con estos predicados y obtener las cláusulas durante una consulta, es vital reemplazar los nombres de las variables por una variable no instanciada.

Además de estas herramientas incorporadas, desarrollar el intérprete en Prolog permite acceder a las funciones básicas del lenguaje. En el intérprete previo, era esencial implementar la función de paso manualmente para cada término especial. Por ejemplo, el predicado `is/2` se usa para evaluar expresiones aritméticas y unificar el resultado con una variable. En la expresión `X is 1 + 2` , `X` se unificará con 3, que es el resultado de $1 + 2$. En el intérprete desarrollado en Python, es necesario implementar una función *pasada* que procese una expresión aritmética y proporcione su resultado, mientras que en esta versión se aprovecha el predicado *is* ya existente.

Otra ventaja de esta implementación es su facilidad de uso. Al estar implementada directamente en Prolog, las herramientas de cobertura están disponibles desde el mismo entorno en el que se programa. Esta integración inmediata es particularmente valiosa en el contexto de los casos de uso, donde se presupone una persona usuaria con experiencia en el entorno Prolog y con interés en una herramienta ágil y accesible.

Sin embargo, esta solución comparte algunas desventajas con el intérprete previo. Aunque utiliza las herramientas de Prolog para lograr un diseño más robusto, todavía necesita desarrollar procedimientos que otros intérpretes ya ejecutan de forma más eficiente. Esto introduce una complejidad adicional en el desarrollo, incrementa la posibilidad de errores y alarga innecesariamente el tiempo de ejecución. Además, si se incorporan nuevas funciones o se realizan cambios específicos en un intérprete, estos tendrán que ser implementados manualmente, lo que eleva el costo de mantenimiento a largo plazo.

En las fases de desarrollo descritas en el apartado 4.4, se considera la incorporación de esta herramienta en el cronograma de trabajo. A pesar de la complejidad inherente al desarrollo de un intérprete completo, ya sea en Prolog o en otro lenguaje, se han empleado dos herramientas existentes para reducir significativamente la carga de trabajo. Para convertir el archivo ".pl" en una serie de tokens clasificados por tipo, se utiliza el módulo *tokenize* [1]. La interpretación de la base de conocimientos a una estructura en la que se pueda realizar la unificación, así como la ejecución de las consultas, se lleva a cabo mediante la herramienta *prologt* [2], con algunas adaptaciones.

Dentro del marco de un *Sprint* empresarial, este trabajo valora de manera positiva la herramienta como una prueba de concepto que podría introducirse antes de desarrollar un prototipo completo. El intérprete Prolog proporciona las funciones descritas en los casos de uso y tiene una extensión de aproximadamente 1200 líneas de código. De estas,

[1] `https://github.com/shonfeder/tokenize`
[2] `https://github.com/mistupv/prologt`

100 líneas se agregaron al código ya existente de *tokenize* y *prologt* para implementar la funcionalidad de cobertura de código.

El desarrollo de este prototipo establece un punto de partida para abordar las deficiencias identificadas y potenciar sus fortalezas. Ofrece una ventaja significativa respecto al prototipo anterior debido a su facilidad de uso y desarrollo. Sin embargo, se beneficiaría de un uso más amplio de las herramientas que proporciona Prolog y un sistema de ejecución más eficiente e integrado.

4.2.3. Instrumentación

A partir del análisis de los intérpretes anteriores, se sugiere la técnica de instrumentación del código para evaluar la cobertura. Esto implica utilizar un intérprete de Prolog ya existente para procesar el programa y ejecutar la consulta. Además, se agregan y modifican hechos y reglas del archivo que se desea medir, con el propósito de recopilar datos acerca de la cobertura durante la ejecución. La herramienta opera previo a la lectura del archivo en Prolog, efectuando cambios al código y, tras finalizar las consultas, recopila la información obtenida y muestra los resultados de la cobertura.

Este método resuelve los problemas de mantenimiento y las restricciones a determinadas funciones a ejecutar que presentaban los prototipos anteriores. Al llevar a cabo la ejecución en un intérprete externo, la herramienta puede admitir los predicados específicos de cualquier intérprete, siempre y cuando estos puedan ser identificados en la fase de instrumentación. Esta característica proporciona a la implementación robustez frente a posibles errores. Si, durante la fase de instrumentación, se identifica un predicado con una estructura no reconocible, este puede ser excluido de la medición de cobertura y permanecer sin cambios, manteniendo el código original intacto.

Dado que la traza de ejecución en las mediciones de cobertura tendrá una estructura similar a una traza de ejecución regular, con llamadas intercaladas a predicados que recolectan datos de cobertura, es posible usar las herramientas de análisis de traza que proporciona el intérprete de forma simultánea. Por ejemplo, la herramienta "trace" de SWI-Prolog, mencionada en la Sección 3.2, podrá emplearse al consultar el código instrumentado para un análisis más profundo.

A pesar de los beneficios que proporciona esta solución en comparación con los intérpretes, presenta también algunas limitaciones. La etapa de instrumentación depende de un tokenizador que, aunque no necesita la misma complejidad que el intérprete mencionado en la Sección anterior, debe tener en cuenta la estructura de las reglas para incorporar los predicados que evalúan la cobertura. Las estructuras específicas de una implementación de Prolog que alteren estos parámetros podrían causar un error en la consulta o limitar el alcance de la cobertura

Por ejemplo, las Gramáticas de Cláusulas Definidas (DCG, por sus siglas en inglés) son una notación usada habitualmente en Prolog (no se encuentra incluida en el estándar ISO pero sí en muchas implementaciones populares) [Pereira and Shieber, 2002]. En lugar de usar el símbolo `:-` para distinguir entre la cabeza y el cuerpo de una regla, las DCG emplean el símbolo `-->`. Para añadir un predicado al cuerpo de una DCG, indicando a Prolog que se debe realizar una consulta sobre este, es necesario rodearlo con `{}`. Consideremos, por ejemplo, un programa que contiene una regla seguida de una DCG:

```
foo :- bar.
bar --> baz.
```

Una instrumentación para añadir el predicado `medir_cobertura` a todas las reglas debería generar el siguiente resultado:

```
foo :- medir_cobertura, bar.
bar --> {medir_cobertura}, baz.
```

A pesar de estas limitaciones, esta solución ofrece un equilibrio favorable entre funcionalidad, resistencia a cambios y facilidad de uso. Presenta una serie de ventajas en comparación con otras soluciones, estableciéndose así como la mejor alternativa para el desarrollo.

4.2.4. Modificación de SWI-Prolog

Como se introdujo en el apartado 3.2, ya existe un método de medición en la implementación de SWI-Prolog. También se ha considerado la posibilidad de tomar este sistema como base y expandirlo para incorporar las funciones deseadas.

La herramienta `test_cover` utiliza una función interna de SWI-Prolog denominada *tracer hooks*. Esta función comprende tablas de información vinculadas a un predicado que cambian mediante el código fuente cuando este se ejecuta. Dichas tablas almacenan información recopilada durante la ejecución, como unificaciones fallidas y exitosas, la cantidad de veces que se consulta el predicado de la base de conocimientos, entre otros datos. Los *tracer hooks* no están incluidos en los estándares ISO del lenguaje; por lo tanto, una herramienta de cobertura basada en estos estaría limitada únicamente a la ejecución en SWI-Prolog.

En la actualidad, esta herramienta no es perfecta y presenta ciertos problemas conocidos. La documentación oficial de `test_cover` señala que: *"Depende en gran medida de las funciones internas de SWI-Prolog. Se ha considerado usar un meta-intérprete para este propósito, pero es muy complejo realizar una meta-interpretación 100 % completa de Prolog. Ejemplos de áreas problemáticas incluyen el manejo de cortes en estructuras de control y llamadas desde meta-predicados no interpretados."*

La documentación detalla también algunas limitaciones presentes en la herramienta:

- *"El rendimiento empeora significativamente (aproximadamente 10 veces más lento).*
- *No es posible utilizar el depurador durante el análisis de cobertura.*
- *Actualmente, la herramienta de análisis de cobertura no es segura en entornos de múltiples hilos."*

Cualquier extensión a la herramienta heredaría estas limitaciones y estaría sujeta a posibles cambios en el funcionamiento de los *tracer hooks*. Esta herramienta, desarrollada en el código fuente de SWI-Prolog, no está documentada y fue diseñada para uso exclusivamente interno.

Se valoró la posibilidad de hacer modificaciones al código interno de SWI-Prolog para implementar la funcionalidad adicional de la herramienta. Esta opción tendría la ventaja de ser la más rápida en ejecución entre las evaluadas, y ofrecería una solución completamente contenida en Prolog, transparente en su uso. Sin embargo, debido a las limitaciones iniciales de los *tracer hooks* como herramienta, considerando la falta de portabilidad de la solución a otras versiones de Prolog y la complejidad técnica de modificar código fuente no documentado en un proyecto del tamaño de SWI-Prolog, se descartó esta solución.

4.2.5. Otras variaciones del problema

Más allá de implementaciones específicas, las soluciones propuestas comparten una serie de características que influyen en el diseño de la herramienta a desarrollar. Además

de evaluar las soluciones por sus ventajas y desventajas, se examinan estas características para determinar las perspectivas que mejor se alinean con la solución deseada.

Desarrollo de un intérprete

Las dos primeras soluciones proponen crear un entorno de Prolog para medir la cobertura, mientras que las siguientes dos se basan en un entorno ya existente y aprovechan sus características para realizar las mediciones.

En este trabajo, se considera más apropiado usar un entorno ya existente, pues se busca maximizar la funcionalidad que se pueda obtener del proyecto. El objetivo es que este pueda medir todas las estructuras presentes en el estándar ISO 13211 y ejecutar predicados específicos de una implementación.

Solución interna o externa a Prolog

Tanto el nuevo intérprete desarrollado en Python como las modificaciones al código de SWI-Prolog, que está implementado en C, implicarían un desarrollo externo al lenguaje Prolog. Sin embargo, este trabajo se centra en implementar una solución interna en Prolog por varias razones.

- Una de las motivaciones de este trabajo es enriquecer la comunidad de Prolog con una nueva contribución y fomentar el desarrollo dentro del lenguaje.
- Prolog, como lenguaje, está especialmente adaptado para crear intérpretes y realizar tareas relacionadas con la compilación, en particular cuando esta compilación es específica para el lenguaje Prolog.
- Como se ha determinado en los casos de uso, las personas usuarias de la herramienta ya contarán con una instalación de Prolog, lo que les permitirá ejecutar cualquier programa de Prolog con facilidad.

Interfaces de la aplicación

Una parte esencial de cualquier software es su interfaz de usuario. En ella deben reflejarse los principios descritos en los casos de uso, el contexto tecnológico en el que opera la herramienta y las características de las personas usuarias.

En los casos de uso presentados en el apartado 4.1.4, se destaca la eficiencia como un atributo clave en el diseño de la aplicación. Esto significa que la aplicación debe reducir al mínimo el tiempo entre su inicio y la ejecución de una consulta que revele la cobertura. Para lograr esto, una interfaz de línea de comandos es la opción ideal, ya que la persona usuaria solo necesita conocer el comando de inicio; al introducirlo en la consola, accederá directamente a los resultados.

En este contexto, una interfaz sería ineficiente si la persona usuaria tuviera que recordar múltiples comandos y sus respectivos parámetros de configuración. Además, dada la naturaleza de la herramienta, que se usa de manera intermitente en el proceso de desarrollo, podría transcurrir bastante tiempo entre el aprendizaje de los comandos y su uso posterior. Esto podría resultar en que la usuaria los olvide y tenga que reaprenderlos.

La principal función de la aplicación es medir la cobertura. Los parámetros de configuración que determinan esta funcionalidad son limitados: qué medidas de cobertura se realizarán y si se hará sobre una única cláusula o varias. Esta simplicidad minimiza las desventajas y favorece el uso de una CLI.

Una interfaz gráfica podría ofrecer ventajas sobre la CLI, al permitir visualizar el archivo original con las medidas de cobertura integradas. Esto presentaría la información de una manera más visual y simplificaría el proceso de aprendizaje del programa. Sin embargo, estas ventajas se ven atenuadas por el requisito de tener un conocimiento inicial considerable de Prolog y la línea de comandos. Por lo tanto, este apoyo visual podría ser limitado en términos de las funciones que ofrece.

Además, en el contexto del desarrollo de código, las personas usuarias de la aplicación ya dispondrán de un entorno de desarrollo personalizado en el que escribirán el código y desde el cual lanzarán la consola. Abrir una nueva interfaz desde esta ya existente puede llegar a ser confuso y molesto para ellas, ya que el manejo de las ventanas interferiría con la eficiencia de su uso. En el caso ideal, la interfaz de desarrollo del código podría tomar los resultados de la cobertura del código e integrarlos en la visualización del fichero que se edita, permitiendo su uso sin dejar la pantalla de desarrollo. Para este aspecto de automatización, o cualquier otro que implique un procesamiento automático de la información resultante de la cobertura, la interfaz CLI ofrece una ventaja significativa, ya que se puede lanzar de forma automática y el formato en texto de sus resultados los hacen idóneos para su procesamiento posterior.

Otro parámetro importante en la consideración de esta característica de diseño es el contexto tecnológico; es decir, el análisis de las interfaces que ofrecen otras herramientas de apoyo para las pruebas en Prolog. Estas herramientas se describen en detalle en el capítulo 3 y se ha observado que siguen principios de diseño similares. Como ejemplo, la herramienta `trace` de SWI-Prolog ofrece una única orden para ejecutar desde la terminal, la cual inicia el proceso de seguimiento de la traza de ejecución. Al lanzar una consulta, esta traza se muestra en la terminal, ofreciendo distintas opciones para seguirla. De manera similar a esta herramienta, en el diseño de `trace`, se prioriza la eficiencia para proporcionar una experiencia de uso rápida y útil.

En conclusión, la elección entre una interfaz gráfica o una ejecución por línea de comandos no es trivial, se ha llevado a cabo una evaluación minuciosa de las necesidades y características relacionadas con la herramienta y su contexto de uso. También se ha tenido en cuenta el entorno tecnológico, los objetivos que persigue y las características de interfaz que emplea para alcanzarlos. Las interfaces existentes en Prolog y su tendencia a favorecer una CLI resaltan la importancia de una solución ágil y directa en este ámbito. La herramienta final deberá, a través de su interfaz, facilitar la integración, ofrecer portabilidad a otros sistemas, permitir el procesamiento automático y proporcionar rutas eficientes para la ejecución.

4.3 Solución propuesta

En el apartado anterior, se han analizado las distintas variaciones del problema tanto desde la perspectiva del desarrollo de una solución final, como desde las características que comparten estas soluciones. El resultado de este análisis, detallado en los apartados previos, es la creación de una herramienta de instrumentación en Prolog para incorporar sentencias de cobertura y obtener las métricas deseadas. A continuación, se enumeran las principales razones que justifican esta decisión, sirviendo como resumen del apartado previo:

- **Portabilidad.** Las medidas se obtendrán como parte del proceso de desarrollo. Por ello, siempre habrá un intérprete de Prolog disponible para ejecutar la solución. Implementar la herramienta en Prolog garantiza que todos las personas usuarias tengan un acceso sencillo a la aplicación.

- **Compatibilidad.** Al instrumentar, la ejecución del programa se externaliza a un intérprete externo, lo que permite que la herramienta de instrumentación omita estructuras que no pueda reconocer. Esto mejora su compatibilidad y la hace más resiliente ante posibles cambios futuros.
- **Eficiencia.** Integrar la herramienta en la consola de Prolog facilita la obtención de resultados de manera más ágil durante su uso, reduciendo interacciones superfluas y tiempos de ejecución.

4.4 Plan de trabajo

Tal como se ha introducido en el apartado 2.4, se utilizará la metodología Scrum para el desarrollo de la herramienta. Esta metodología se centra en ofrecer un desarrollo basado en la transparencia, la inspección y la adaptación [Schwaber and Beedle, 2001]. A continuación, se analizan las características del problema desde la perspectiva del desarrollador para determinar cómo se ajusta al enfoque Scrum.

4.4.1. Principios Scrum

Transparencia

El principio de transparencia en la metodología Scrum sostiene que tanto los procesos como el trabajo realizado en el proyecto deben ser visibles tanto para quienes llevan a cabo el desarrollo como para quienes reciben el producto. Este principio asegura que no se tomen decisiones sobre características de la herramienta con poca transparencia, ya que estas pueden tener una mayor probabilidad de reducir el valor e incrementar el riesgo de fallo.

En el contexto de la herramienta que se está desarrollando en este proyecto, la transparencia podría parecer un aspecto menor, dado que el alumno actúa como el equipo de desarrollo y los resultados de cada iteración se comparten solo con el tutor. Sin embargo, en el proceso de creación de una herramienta destinada a medir la cobertura del código, la transparencia es un principio esencial en el diseño. Considerando que las personas usuarias son desarrolladoras con un nivel técnico avanzado, ellas poseen la capacidad de hacer ajustes en la herramienta para mejorarla y adaptarla a sus necesidades específicas. Para facilitar este proceso, es crucial que la herramienta final tenga un diseño transparente, lo cual se puede asegurar más eficazmente a través de un proceso de desarrollo transparente.

Inspección

El principio de inspección en la metodología Scrum sostiene que se debe revisar frecuentemente el progreso hacia los objetivos trazados para identificar posibles problemas o desviaciones. Esta inspección está estrechamente ligada al principio de adaptación, dado que el propósito de dicha revisión es ajustar dinámicamente el desarrollo ante las adversidades y cambios que se presenten.

El principio de inspección desempeña un papel fundamental en el desarrollo de esta herramienta. A pesar de que el objetivo final de medir la cobertura de código en Prolog es claro y directo, la inherente complejidad del lenguaje y las distintas variaciones al plantear el problema originan múltiples puntos de error y enfoques incorrectos. Desde el punto de vista académico, es importante resaltar que, antes de iniciar el desarrollo de esta

herramienta, mi experiencia programando en Prolog era de apenas 50 líneas de código. Por ello, el proceso de revisión e inspección a lo largo de todo el desarrollo adquiere aún mayor relevancia, ya que facilita la adquisición de conocimientos y permite atender posibles fallos en el diseño.

Adaptación

El principio de adaptación en la metodología Scrum sigue al de inspección. Si durante la inspección se detecta una desviación de los objetivos o un resultado incorrecto, es necesario realizar los ajustes adecuados para mantener una dirección clara y un rumbo definido.

La importancia del principio de inspección en la ejecución de este proyecto resalta la necesidad de adaptación. Como se describe en la Sección 4.2, se proponen varias soluciones viables para cada aspecto del problema. A pesar de haber elegido un enfoque específico para el desarrollo, este podría requerir ajustes que contemplen otras perspectivas, o incluso podría beneficiarse de la exploración de soluciones intermedias basadas en diferentes principios. Estos ajustes, que podrían surgir debido a necesidades identificadas durante el proceso de desarrollo o limitaciones en la implementación, deben implementarse lo antes posible para minimizar cambios en la planificación y desviaciones de los objetivos.

4.4.2. Definición del *Sprint*

Para definir las tareas y fases del proyecto, es esencial comprender el marco temporal que establece el método Scrum. La unidad principal de este desarrollo es el *Sprint*. Cada *Sprint* tiene una duración fija, menor a un mes; para este proyecto, se ha optado por dos semanas. Al concluir este periodo, finaliza el *Sprint* en curso y da inicio el siguiente. Todos los eventos previstos en la metodología se desarrollan dentro de cada *Sprint*, estableciendo así un desarrollo cíclico que se puede seguir fácilmente sin necesidad de un cronograma complejo.

Cada *Sprint* persigue un objetivo específico. Todo el trabajo efectuado durante el *Sprint* debe contribuir a ese objetivo. Se definen los propósitos de cada *Sprint* y se calcula el número de *Sprint*s necesarios para concluir el proyecto desde su inicio. Sin embargo, la metodología Scrum sigue los principios ágiles [Abrahamsson et al., 2017], que promueven una estructura iterativa, permitiendo adaptar los requisitos si es necesario para lograr el objetivo final. Así, las fases de desarrollo descritas en esta Sección podrían sufrir modificaciones para llegar a la solución final, lo cual no significa un error en la aplicación del método Scrum.

Los *Sprints* incluyen también una serie de eventos asociados, entre los que se encuentran la planificación del *Sprint*, el Scrum diario, la revisión del *Sprint* y la retrospectiva del *Sprint*. En el contexto del método Scrum, estos eventos involucran a equipos donde distintas personas cumplen diferentes roles. Sin embargo, en el marco de este proyecto, solo se considera la intervención del estudiante y el tutor en el proceso de desarrollo. Por ello, es crucial adaptar estos eventos para que se adecuen al carácter de un trabajo de fin de grado.

Planificación del *Sprint*

En la metodología Scrum, la Planificación del *Sprint* es el evento que da inicio a un *Sprint*. Durante este evento, se exploran las tareas a desarrollar en el *Sprint*, valorando

su aporte al objetivo final, así como la estrategia que el equipo empleará para llevarlas a cabo. El propósito principal de este evento es definir los elementos clave del próximo *Sprint*, estableciendo tareas específicas orientadas a un objetivo claro que guiará las acciones del equipo. En relación con este proyecto, la planificación del *Sprint* no requiere grandes cambios; se llevará a cabo una reunión entre el tutor y el alumno antes de empezar el *Sprint* para definir los objetivos y tareas a realizar.

Scrum Diario

El Scrum Diario es un evento que dura 15 minutos destinado al equipo de desarrollo. Durante este tiempo, se revisa el avance hacia el objetivo del *Sprint* y se analizan las tareas del día. El propósito principal del Scrum Diario es la comunicación, para garantizar que todo el equipo esté al tanto de la dirección del proyecto y que el trabajo que se realiza persigue un objetivo común. En este proyecto particular, es necesario hacer una adaptación significativa, dado que el equipo de desarrollo consiste en una sola persona y el avance no es constante a lo largo del *Sprint*.

La adaptación propuesta en este documento sugiere establecer entre 7 y 10 incrementos específicos en cada *Sprint*. La finalidad es realizar un seguimiento detallado del desarrollo del *Sprint*, de modo que incluso las variaciones estén planificadas en incrementos bien definidos. Al finalizar cada uno de estos incrementos, se efectúa una revisión tanto de los logros alcanzados como de los pendientes para llegar a la meta. Basándose en esta revisión, se hacen las correcciones necesarias en los incrementos restantes.

Revisión del *Sprint*

El objetivo de la Revisión del *Sprint* es doble. Por un lado, se realiza un análisis de los resultados del *Sprint* tanto desde una perspectiva funcional como en relación con el objetivo general. A partir de estos resultados, se sugieren ajustes para enfrentar cualquier problema detectado o cambio en el entorno. Este evento no es únicamente una exposición de los resultados, sino que se debe llevar a cabo una evaluación activa donde se plantean alternativas para aplicar en iteraciones futuras. En este proyecto, la revisión del *Sprint* se efectúa entre el tutor y el alumno, siguiendo una dinámica similar a la establecida por la metodología Scrum. Para optimizar el tiempo, esta revisión se realiza en una única reunión, que también incluye la planificación del siguiente *Sprint*, lo que permite reducir el número de reuniones a una por cada *Sprint*.

Retrospectiva del *Sprint*

El objetivo principal de la Retrospectiva de *Sprint* es identificar y planificar formas de mejorar la calidad del trabajo y la eficacia del equipo. Durante esta reunión, el equipo de desarrollo examina tanto su trabajo como sus comunicaciones e interacciones, buscando identificar aspectos que han influido tanto positiva como negativamente en el desarrollo del proyecto. A partir de este análisis, se determinan las modificaciones más efectivas para potenciar la eficacia del equipo y se implementan en la organización del siguiente *Sprint*.

En este proyecto, la Retrospectiva de *Sprint* se realiza en la misma reunión donde se lleva a cabo la revisión y planificación del *Sprint*. Durante este tiempo, se revisan los procesos y herramientas utilizados en el proyecto. Estos elementos pueden variar desde suposiciones iniciales que han cambiado a lo largo del desarrollo, hasta herramientas que podrían ser reemplazadas por alternativas más efectivas. Además, en este evento se

explora el trabajo realizado y las tareas pendientes desde una perspectiva académica. Por ejemplo, se considera la posibilidad de investigar alternativas emergentes con el objetivo de obtener una comprensión más amplia del problema abordado.

4.4.3. Fases de desarrollo

Sprint	**Nombre**	**Objetivos**
1	Prueba de concepto	Utilizar el intérprete existente para medir la cobertura de código. Este deberá: ▪ Mostrar información sobre la cobertura resultante de una consulta ▪ Medir la cobertura de todas las sentencias del programa según la consulta proporcionada
2	Instrumentador básico	Desarrollar una herramienta de instrumentación que mida la cobertura de código y muestre los resultados. Esta deberá: ▪ Manejar un amplio subconjunto del lenguaje Prolog, admitiendo operaciones tales como `'cut'`, `'fail'` y `'call'`. ▪ Ofrecer un método de ejecución accesible y sencillo desde la consola de Prolog.
3	Instrumentador modular	Expandir la herramienta de instrumentación para incluir diferentes modos de medición. En este *Sprint*, la herramienta deberá: ▪ Establecer los fundamentos para añadir otras opciones de medida sin necesidad de reestructurar el código. ▪ Mostrar información adicional sobre la cobertura, incluyendo métricas avanzadas como la unificación con las cláusulas de un predicado y la instanciación de variables en la consulta.

4	Interfaz	Integrar opciones para tablas personalizables en la visualización de cobertura, diseñar una interfaz de consola y permitir la ejecución de múltiples consultas. La herramienta deberá: ▪ Facilitar una ejecución sencilla y configurable, permitiendo ajustar los resultados mostrados después de una consulta. ▪ Habilitar la ejecución directa desde la consola de Prolog y también mediante línea de comando. ▪ Admitir la ejecución de múltiples consultas mediante un archivo ".plt".

Como se ha establecido en la Sección anterior, cada *Sprint* estará compuesto por una serie de Incrementos definidos que marcarán el avance hacia el objetivo en cada iteración. A continuación, se presenta una predicción aproximada de estos incrementos, con la finalidad de ofrecer un mayor detalle sobre las tareas a completar en el proyecto. Debido a la naturaleza de los incrementos, es probable que estos cambien significativamente durante el desarrollo. Su propósito principal no es detallar las tareas específicas del proyecto, sino mostrar un avance claro para reforzar la visión general de los objetivos establecidos.

Pese a esta incertidumbre, este desglose de tareas se propone para lograr una visión más clara del trabajo a realizar. Al abordar el proyecto desde las perspectivas de iteraciones a completar en un periodo determinado y de tareas específicas a llevar a cabo, se puede conseguir una evaluación más precisa del esfuerzo necesario y hacer una predicción más fundamentada de los recursos requeridos.

Como se establece en la guía PMBOK [Project Management Institute, 2021], estos desgloses en contextos cambiantes, que a primera vista podrían parecer irrelevantes, empujan al equipo de desarrollo a considerar nuevas perspectivas del problema y a profundizar su comprensión de sus características.

Sprint	**Índice**	**Incremento**
S1	I1	Lectura de la biblioteca `'tokenize'`.
	I2	Lectura del procesador de base de conocimientos de la biblioteca `'prologt'`.
	I3	Lectura de la ejecución de una consulta de `'prologt'`.
	I4	Implementación de la función para guardar la cobertura durante una consulta.
	I5	Implementación de cálculos de cobertura por predicado.
	I6	Integración final de las partes, creando una versión funcional.
	I7	Pruebas del sistema.
S2	I1	Desarrollo de un sistema de instrumentación básico para añadir un predicado `'flag'` a cada sentencia.
	I2	Desarrollo de un sistema para pasar argumentos en la instrumentación.

	I3	Implementación de un registrador `'logger'` para manejar las llamadas durante la consulta.
	I4	Implementación de un visualizador para mostrar los resultados de la cobertura.
	I5	Segmentación de la instrumentación en módulos: instrumentación, procesamiento de consultas y visualización.
	I6	Implementación de predicados para admitir operaciones `'call'`, `'cut'` y `'fail'`.
	I7	Implementación de una interfaz de predicados para realizar la ejecución desde la consola.
	I8	Pruebas del sistema.
S3	I1	Creación de un módulo `'launcher'` para gestionar una instrumentación más avanzada.
	I2	Implementación de configuraciones para el instrumentador.
	I3	Implementación de la opción `'branch'` para medir la unificación con las cláusulas del predicado.
	I4	Implementación de la opción `'ground'` para medir la instanciación de variables.
	I5	Modificaciones al módulo `'logger'` para recopilar y almacenar información de las nuevas opciones.
	I6	Modificación del módulo de visualización para mostrar los resultados.
	I7	Pruebas del sistema.
S4	I1	Desarrollo del módulo de visualización para permitir la configuración de la tabla mostrada.
	I2	Implementación de medidas especiales en el módulo de visualización, como acumular cobertura por predicado .
	I3	Implementación de un módulo `'plcov'` para ofrecer un único punto de acceso a la herramienta.
	I4	Desarrollo de una herramienta bash para ejecutar el software desde la línea de comandos.
	I5	Desarrollo de un módulo `'loader'` para gestionar la carga de archivos con consultas y la limpieza de datos después de las mediciones.
	I6	Desarrollo de una suite de pruebas (integrando las pruebas previas).
	I7	Generación de archivos de documentación del código en el entorno PlDoc.

4.4.4. Riesgos

Para realizar un análisis de riesgos, utilizamos algunas de las herramientas propuestas en la guía PMBOK. Al adaptar las categorías y subcategorías de riesgo de la guía PMBOK al contexto de este proyecto, obtenemos la Estructura de Desglose de Riesgo (RBS, por sus siglas en inglés), que se muestra en la Figura 4.4.

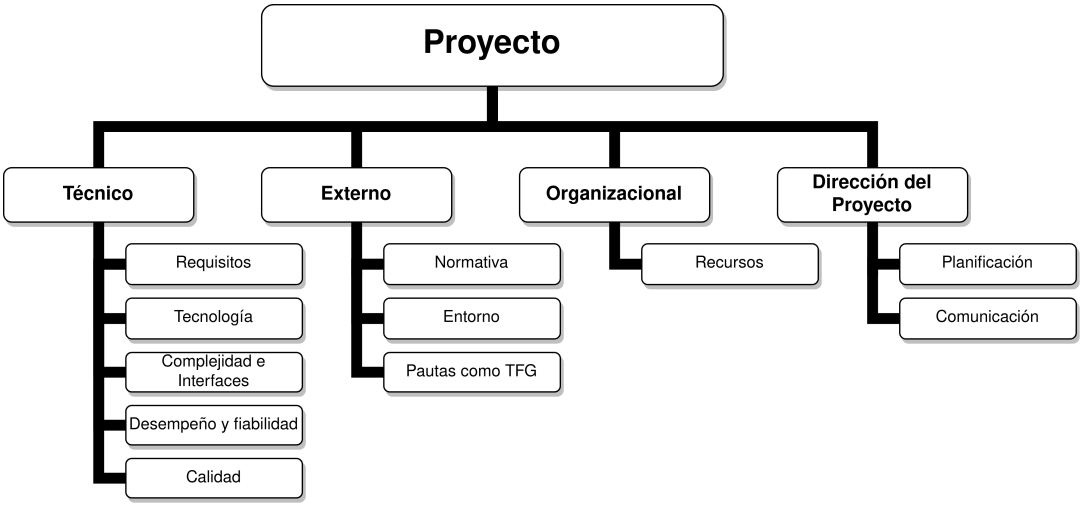

Figura 4.4: Estructura de Desglose del Riesgo para el proyecto.

A continuación se muestra una tabla con algunos riesgos identificados para el proyecto, categorizados por categoría y subcategoría:

Categoría	**Subcategoría**	**Riesgo**
Técnico	Desempeño y Fiabilidad	Rendimiento insatisfactorio del sistema bajo cargas de trabajo elevadas.
Técnico	Calidad	Falta de modularidad en el código que pueda impedir añadir nuevas funcionalidades al programa o adaptar a cambios en el entorno fácilmente.
Organizacional	Recursos	Impacto en los plazos de la metodología Scrum debido a limitaciones de tiempo por otras tareas.
Dirección del Proyecto	Planificación	Desviaciones en la planificación a causa de una estimación errónea del tiempo necesario para aprender un nuevo entorno.

En el manual PMBOK se establece una serie de documentos posteriores a la estructura de Desglose del Riesgo con el objetivo de evaluar la probabilidad e impacto asociados a cada riesgo. Este enfoque integra diversos aspectos del proceso de dirección de proyectos, incluyendo costes, duración e importancia de las actividades involucradas. También se analiza cómo un riesgo puede afectar un cronograma ya establecido y de qué manera la gestión del riesgo puede influir en los costes del proyecto.

Para ofrecer una mayor claridad en el proceso de análisis de riesgos, se detalla uno de los riesgos presentados en la tabla: ”Desviaciones en la planificación debido a una estimación incorrecta del tiempo requerido para aprender un nuevo entorno”. Este riesgo es significativo en el plan de trabajo propuesto, dado que se utilizan múltiples herramientas y estrategias que no son conocidas previamente. El tiempo necesario para dominar estas herramientas podría subestimarse, especialmente porque, al momento de planificar, los detalles específicos de la herramienta aún son desconocidos. Al concluir el análisis de este riesgo, se examina el impacto potencial en el proyecto y las estrategias para mitigarlo.

Impacto al Proyecto. El riesgo puede tener un impacto significativo, ya que el aprendizaje de un nuevo entorno afecta a las tareas que dependen de dicho entorno. Además, los entornos seleccionados son esenciales para la ejecución del proyecto, lo que hace difícil adaptar la metodología a una alternativa.

Estrategias de mitigación. Para reducir este riesgo, se sugiere realizar una estimación ponderada que incluya un margen de tiempo adicional para el aprendizaje de las herramientas. Esta estimación puede ser ajustada en función de las herramientas que se hayan aprendido hasta ese momento, garantizando así que la planificación se vuelva más precisa a medida que el proyecto progresa.

La metodología Scrum propone una estructura de tareas y hitos menos estricta que una metodología tradicional de gestión de proyectos [Hammad and Inayat, 2018]. En esta Sección, se presenta una estructura general con el objetivo de contextualizar los riesgos. Esta estructura tiene como finalidad realizar una revisión durante el evento de Retrospectiva de cada *Sprint*, para identificar posibles riesgos de la siguiente iteración.

Para incorporar un margen en el proceso de desarrollo del proyecto y enfrentar adecuadamente los posibles riesgos en el cronograma, se lleva a cabo una evaluación independiente de la duración del proyecto. Para esta evaluación, se utiliza la técnica de estimación por tres puntos, desglosando las horas desde un enfoque analítico. Este proceso se expone en el apartado 4.5 como herramienta principal para establecer el presupuesto.

4.4.5. Marco Legal

El lenguaje de programación Prolog en sí mismo, como sistema de inferencia lógica, no tiene una licencia específica asociada. Prolog es un concepto general y una especificación de lenguaje que define una manera de realizar cálculos simbólicos basados en principios de programación lógica. Sin embargo, las distintas implementaciones de Prolog, como SWI Prolog, GNU Prolog y otras, sí tienen licencias que dictan cómo se pueden usar, distribuir y modificar dichas implementaciones.

SWI Prolog se distribuye bajo la Licencia Pública General Menor (LGPL, por sus siglas en inglés) [Free Software Foundation, 2007], que permite la modificación y distribución del software manteniendo las mismas libertades en las versiones modificadas. El programa desarrollado requiere la interpretación de SWI-Prolog para realizar su ejecución pero no distribuye a la herramienta.

La herramienta desarrollada se adhiere al código libre, un modelo de licencias software que concede permisos de distribución, modificación y uso. El marco de código libre se basa en la libertad de acceso a la información y el conocimientos para que así estos puedan beneficiar a cualquier persona [Stallman, 2002].

4.5 Presupuesto

La herramienta desarrollada se adhiere al paradigma del software libre. El objetivo del proyecto es contribuir a una comunidad y mejorar aspectos del proceso de desarrollo en Prolog, enfocando el proyecto como una herramienta y no simplemente como un "producto". No obstante, se ha incorporado esta Sección para evaluar la carga de trabajo y el esfuerzo requerido como un componente esencial del desarrollo.

Durante el desarrollo del proyecto, se emplea el software SWI-Prolog. Su licencia LGPL, descrita en el apartado 4.4.5, permite su uso de manera gratuita, por lo que no representa ningún costo adicional.

La duración del desarrollo puede ser analizada desde dos niveles de detalle. Por un lado, mediante la metodología Scrum se planifican 4 *Sprints* de dos semanas cada uno. Esta planificación se plantea desde la perspectiva de un trabajo final de grado que se lleva a cabo en paralelo con otras actividades y presupone un tiempo de desarrollo de hasta

dos horas diarias. Al multiplicar este tiempo por los 4 *Sprints*, se estima un esfuerzo total de aproximadamente 100 horas, excluyendo el aprendizaje del lenguaje y el desarrollo de las soluciones alternativas, que suman unas 200 horas añadidas.

Para obtener un cálculo más detallado, se puede realizar un análisis del desglose de incrementos presentados en el apartado 4.4.3. Desde esta perspectiva, se consideran tareas específicas en una secuencia temporal, por lo que es viable utilizar la Técnica de Revisión y Evaluación de Proyectos (PERT, por sus siglas en inglés) [Taylan et al., 2020]. Este método, frecuentemente utilizado en proyectos de desarrollo de software, considera un escenario optimista O, uno más probable M y uno pesimista P para calcular la duración estimada de la tarea, denominada tiempo esperado, con la fórmula $TE = \frac{O+4*M+P}{6}$. Esta técnica pone énfasis en el escenario más probable, pero también asume que los escenarios optimista y pesimista pueden materializarse con igual probabilidad. La tabla 4.7 muestra el desglose de los incrementos de cada *Sprint*, junto con sus estimaciones temporales.

Índice	Optimista	Probable	Pesimista	Esperado
S1-I1	2	4	6	4.0
S1-I2	1	3	5	3.0
S1-I3	1	2	4	2.2
S1-I4	2	3	5	3.2
S1-I5	3	4	7	4.3
S1-I6	4	6	8	6.0
S1-I7	2	3	5	3.2
Total *Sprint* 1	**15**	**25**	**40**	**25.8**
S2-I1	1	2	3	2.0
S2-I2	2	3	4	3.0
S2-I3	1	2	3	2.0
S2-I4	2	3	4	3.0
S2-I5	3	4	6	4.2
S2-I6	4	5	7	5.2
S2-I7	2	3	4	3.0
Total *Sprint* 2	**15**	**22**	**31**	**22.3**
S3-I1	1	2	3	2.0
S3-I2	1	2	4	2.2
S3-I3	2	3	5	3.2
S3-I4	2	4	6	4.0
S3-I5	3	5	7	5.0
S3-I6	4	6	8	6.0
S3-I7	2	3	5	3.2
Total *Sprint* 3	**15**	**25**	**38**	**25.5**
S4-I1	1	2	3	2.0
S4-I2	2	3	4	3.0
S4-I3	1	2	4	2.2
S4-I4	2	3	5	3.2
S4-I5	3	4	6	4.2
S4-I6	4	5	7	5.2
S4-I7	2	3	4	3.0
Total *Sprint* 4	**15**	**22**	**33**	**22.7**
Total Proyecto	**60**	**94**	**142**	**96.3**

Tabla 4.7: Estimación de tiempo invertido por Incremento en cada *Sprint* utilizando PERT.

El tiempo total, al considerar las tareas específicas dentro de cada *Sprint*, es de 96 horas, lo que coincide aproximadamente con la suma de las semanas de los 4 *Sprints*. Esto respalda la predicción y fortalece la confianza en que se cumplirá. Asimismo, es destacable que la duración de las tareas dentro de cada *Sprint* varía conforme avanza el proyecto: los *Sprints* iniciales requieren más tiempo, mientras que los *Sprints* finales son más breves. A pesar de que esta distribución podría no coincidir con la duración constante de las iteraciones Scrum, las tareas propuestas para cada *Sprint* son flexibles y pueden trasladarse al *Sprint* siguiente. De este modo, el espacio disponible en los *Sprints* finales proporciona un margen para tareas adicionales y posibles problemas que surjan durante el desarrollo.

CAPÍTULO 5
Diseño de la solución

5.1 Arquitectura del sistema

El desarrollo de la herramienta se realizará principalmente mediante el lenguaje Prolog. Dada la naturaleza de Prolog como un lenguaje lógico puro, no es posible traducir directamente los principios de diseño que se aplican en otros lenguajes, ya sean imperativos u orientados a objetos. A pesar de ello, sigue siendo esencial tener en cuenta los patrones arquitectónicos empleados en el desarrollo y el marco que define las diversas tareas y relaciones del programa. De esta forma, se puede asegurar un código limpio y bien estructurado.

En cualquier código de Prolog, se emplean dos estructuras principales para establecer la jerarquía del programa: los módulos de Prolog y los predicados específicos definidos en cada módulo. En un programa, los módulos funcionan como herramientas de encapsulación de información. Estos módulos ofrecen funciones implementadas a través de predicados internos, los cuales son accesibles mediante predicados públicos disponibles para otros módulos.

En la Figura 5.1, se presentan los diferentes módulos que componen la herramienta, dispuestos según el flujo de ejecución. La interacción con la persona usuaria se realiza a través del módulo *plcov*, que administra tanto el procesamiento del fichero como el lanzamiento de consultas posteriores.

Durante el procesamiento del fichero, el módulo *plcov* invoca al módulo *loader*. Este último localiza el fichero que contiene el programa y envía sus predicados al módulo *instrumenter*. Este módulo introduce una sentencia de cobertura en los predicados del fichero, generando así una lista de predicados instrumentados. Además, el módulo *instrumenter* recaba información sobre los predicados del fichero, asignándoles un identificador y definiendo su tipo (ya sea regla, sentencia, DCG, entre otros).

PRespecto a las consultas, el módulo *plcov* consulta el programa instrumentado, previamente almacenado y cargado en Prolog desde el fichero *".temp.pl"*. Los predicados instrumentados en *.temp.pl* hacen llamadas al módulo *logger*, responsable de registrar y acumular los valores de cobertura del programa. Una vez concluida la llamada al predicado, *plcov* activa al módulo *loader*, el cual recoge los resultados de cobertura de la consulta y se los envía al módulo *visualizer* para mostrar la cobertura final.

El módulo *utils* alberga predicados definidos de manera general, que son utilizados por todos los otros módulos para llevar a cabo funciones descritas en la Sección 6.2.3.

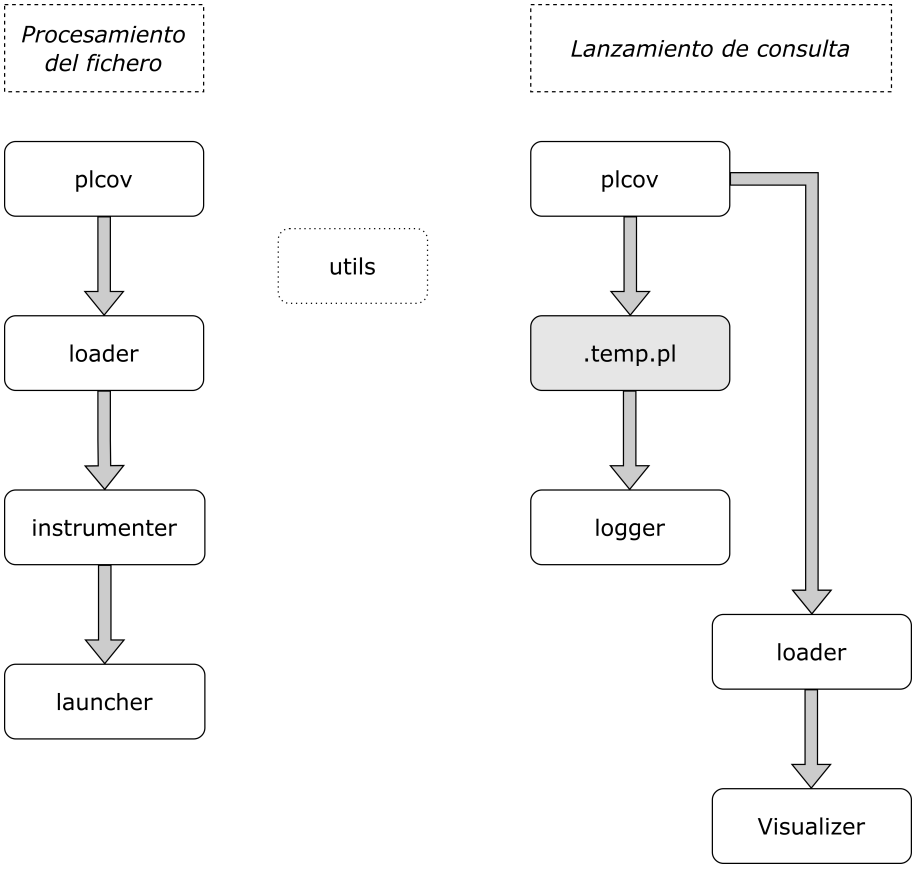

Figura 5.1: Módulos de la herramienta, conectados por sus dependencias

Dentro de un módulo particular, se sigue la estructura que se muestra a continuación para nombrar los predicados:

- En primer lugar, el archivo de un módulo se organiza según los predicados exportados, siguiendo el orden especificado en la lista de predicados a exportar.
- Cada Sección contiene el desarrollo de la lógica correspondiente al predicado exportado. Dicho predicado puede incluir múltiples predicados adicionales.
- Los predicados privados de una Sección se listan en el orden en que aparecen como condiciones en los predicados definidos previamente.

5.2 Diseño Detallado

En el libro "Clean Code" [Martin, 2008], se establecen diversos principios, indicaciones y recomendaciones para escribir código limpio. El propósito del código limpio es desarrollar un programa que sea fácil de comprender; esto significa que el código debería poder ser leído, modificado, extendido y mantenido, incluso por una persona externa al equipo original del proyecto.

Aunque estas directrices se contemplan principalmente desde una perspectiva de desarrollo imperativo y orientado a objetos, no significa que no puedan ser adaptadas a un entorno de programación declarativa. Durante el desarrollo de esta herramienta, se categorizan los aspectos del código limpio en diferentes secciones para la implementación. El propósito es investigar las adaptaciones necesarias para un contexto declarativo, así como su consideración en el prototipo final.

La división considera cuatro aspectos esenciales del código:

- **Diseño:** abarca los componentes clave del código para asegurar un diseño modular y flexible.
- **Estructura:** esta categoría subraya la organización y el formato del código, buscando proporcionar una estructura visualmente clara y legible.
- **Código:** esta Sección se centra en la creación de código con una lógica transparente, legibilidad y mínima complejidad.
- **Nombres y comentarios:** este apartado se refiere a la selección de nombres para funciones y variables, y a la inclusión de comentarios pertinentes en el código.

A continuación se describen con más detalle las directrices establecidas por el libro "Clean Code" para cada aspecto. Para cada aspecto se describen también las adaptaciones consideradas para el entorno declarativo, acompañadas de ejemplos de la herramienta final para ilustrar su aplicación.

5.2.1. Diseño

La categoría de diseño se centra en aspectos fundamentales de la organización del código. Algunas de las recomendaciones que se presentan en esta Sección son:

- Mantener las configuraciones variables en niveles altos. Esto permite ajustar el comportamiento del programa sin alterar la lógica subyacente, y asegura que los efectos de la configuración sean coherentes con su relevancia.
- Optar por el polimorfismo en lugar de if/else o switch/case. Esta aproximación transfiere parte de la lógica a la estructura de los procedimientos, simplificando su comprensión y reduciendo las condiciones que el programador debe gestionar.
- Ocultar la estructura interna. Al encapsular las funciones de tal manera que su lógica interna sea transparente, se abstrae la implementación de su uso. Esto limita las dependencias, mejora la comprensión del código y lo hace más extensible.
- Evitar estructuras híbridas (mitad objeto, mitad datos). Este principio está estrechamente relacionado con los lenguajes orientados a objetos. Un objeto ofrece funcionalidades que se pueden acceder desde el exterior, pero no permite el acceso a su funcionamiento interno. Por otro lado, una estructura de datos proporciona acceso a su contenido pero no ofrece funciones aplicadas al contexto de su uso. Las estructuras híbridas no son recomendables porque permiten el acceso a su estructura interna, lo que complica su modificación debido a posibles consecuencias imprevistas en otras partes del programa. A su vez, al ofrecer funcionalidad externa, cambiar su operación también puede llevar a resultados inesperados.

Para adaptar estas recomendaciones al entorno de Prolog, es necesario tener una comprensión más profunda de la relación entre la lógica y los datos en el paradigma declarativo. En un programa imperativo, la lógica se define mediante las instrucciones dadas, las cuales pueden generar información representada en estructuras de datos y modificar esa información a través de operaciones predeterminadas. El programa escrito en este caso representa un proceso definido.

Por otro lado, en un programa declarativo, las instrucciones representan una serie de declaraciones. Estas declaraciones pueden estar vinculadas a estructuras de datos o hacer

referencia a operaciones predeterminadas. En este contexto, el programa escrito no representa un proceso per se, sino un conjunto de sentencias que pueden ser seleccionadas en un orden particular para resolver un problema planteado.

En el paradigma imperativo, la programación orientada a objetos organiza las instrucciones del programa en unidades o clases, que a su vez pueden albergar estructuras de datos [Rentsch, 1982]. Estas unidades o clases pueden encapsular funciones y datos, ofreciendo una estructura que refleja las diferentes dimensiones del problema de manera más intuitiva.

Las recomendaciones de diseño tienen como objetivo reducir la complejidad de las instrucciones de un programa. Entre las recomendaciones propuestas, existen diversas estrategias para disminuir la complejidad. Por ejemplo, se puede optimizar la relación entre los datos de un programa y las instrucciones que se les aplican, ya sea mediante la traslación de lógica al polimorfismo, reflejando diferentes condiciones en la estructura del código, o minimizando las dependencias entre instrucciones para establecer divisiones más claras en el código.

Con esta divisiones, se pueden adaptar las recomendaciones de código limpio para aplicarlas a un lenguaje declarativo y a la herramienta:

- Módulos independientes y claros. Los módulos del programa deberán representar una parte específica de la ejecución de manera coherente, utilizando predicados públicos que describan su función.
- Favorecer cláusulas individuales sobre condicionales. Aunque Prolog dispone de herramientas que pueden emular los efectos de estructuras condicionales, estas dificultan la lectura declarativa. Estos efectos pueden representarse de forma más natural mediante múltiples cláusulas.
- Distinguir entre argumentos determinados y no determinados en las reglas. Los argumentos de una regla deben categorizarse como determinados o no determinados, y no deben permitir variaciones, pues estas generan incertidumbre en las consultas y dificultan la detección de errores.
- Gestionar las variaciones de configuración en niveles superiores. Similar a la recomendación anterior, las variaciones de configuración deben resolverse al comienzo del proceso, idealmente en un módulo distinto al encargado de la búsqueda de soluciones.

Como se ha descrito, los lenguajes declarativos no siguen el orden escrito de las sentencias en sus programas. El intérprete selecciona los predicados del programa en Prolog que se unifican con una consulta, y consulta repetidamente la base de conocimientos para resolver sus condiciones. Durante estas consultas, el intérprete decide qué cláusulas seleccionar para la unificación. Si existen múltiples cláusulas entre las cuales elegir, se crea un punto de decisión que se explorará si la cláusula seleccionada resulta errónea.

Prolog ofrece una serie de herramientas que permiten interactuar directamente con la búsqueda. Estas herramientas no tienen equivalente en lenguajes imperativos, por lo que es interesante considerarlas desde una perspectiva de código limpio. En la programación declarativa pura, el proceso de búsqueda de solución se abstrae por completo del individuo que escribe la base de conocimientos. Por lo tanto, siempre que sea posible, es preferible evitar el uso de estas estructuras.

A pesar de ello, en ocasiones es necesario utilizar herramientas para controlar la búsqueda de posibilidades según la lógica del programa. En otros momentos, estas herramientas pueden servir para evitar la exploración innecesaria de resultados. Cuando se

limita parte de la búsqueda sin cambiar la solución final, esto se conoce como ”corte verde”. Estos cortes verdes deben incorporarse después del desarrollo inicial de la herramienta y es esencial usar un conjunto de pruebas para garantizar que el comportamiento del programa permanece inalterado.

Los cortes que modifican la búsqueda de la solución para cambiar el resultado final se denominan ”cortes rojos”. Estos cortes son peligrosos ya que rompen la naturaleza declarativa del programa y pueden generar errores difíciles de resolver. En la mayoría de los casos en los que se requiere un corte rojo, este sirve para distinguir cuando la base de conocimientos *no* contiene una sentencia. Por ejemplo, en la siguiente base de conocimientos:

```
color(verde).
color(azul).
color(amarillo).
```

Si se desea introducir un predicado `no_es_color/1` que sea cierto cuando su argumento no es un color, se escribiría de la siguiente manera:

```
no_es_color(X) :- color(X), !, fail.
no_es_color(_).
```

En este ejemplo, se han utilizado dos funciones de Prolog que afectan la búsqueda de soluciones. La primera cláusula del predicado `no_es_color/1` contiene tres términos en su cuerpo. El primer término busca una cláusula del predicado `color/1` en la base de conocimientos; es decir, verifica si el argumento es un color. El segundo término, `!/0`, denominado ”corte”, restringe la búsqueda de soluciones y elimina el punto de decisión para el predicado en el que se encuentra. Esto significa que indica al intérprete que ignore las demás cláusulas de `no_es_color/1` y trate la actual como la única alternativa. Antes de ejecutar este ”corte”, el intérprete conservaba la opción de explorar la segunda cláusula de `no_es_color/1`, opción que habría tomado si la primera cláusula hubiera fallado. El tercer término, `fail/0`, le indica al intérprete que la solución actual no es válida, es decir, este predicado es equivalente a una consulta que resulta en fallo sobre la base de conocimientos.

Finalmente, si la instancia del argumento `X` no se halla en la base de conocimientos, la primera cláusula falla antes de llegar al segundo hecho, eliminando así la posibilidad de probar con la segunda cláusula. Por consiguiente, el intérprete intenta unificar con esta segunda cláusula, la cual siempre logra una unificación exitosa.

Debido a la falta de transparencia en este enfoque de diseño de predicados y la infracción de los principios del paradigma declarativo, Prolog proporciona un método más estructurado para llevar a cabo esta tarea: el predicado `not/1`. Usando este predicado, `no_es_color/1` se podría expresar de la siguiente forma:

```
no_es_color(X) :- not(color(X))
```

A pesar de la existencia del predicado `not/1` y herramientas como el corte, en relación con los principios de un código limpio, se recomienda limitar al máximo el uso de cortes rojos. Si es necesario emplearlos, deben ir acompañados de un comentario que describa su propósito y las consecuencias en el código.

5.2.2. Estructura

Las recomendaciones estructurales se refieren al orden y formato de las líneas de código en un programa, ya sea dentro de un módulo específico o fuera de él. Destacan las siguientes recomendaciones estructurales:

- Funciones similares deberán estar cerca. Esto facilita la lectura del archivo y proporciona una claridad adicional sobre la relación entre diferentes funciones.
- Las líneas de código deberían ser cortas. Esto favorece la lectura vertical del código y refuerza el principio de minimizar su complejidad.
- Mantener una estructura de indentación. Esto organiza el código y permite visualizar sus distintas secciones sin la necesidad de herramientas adicionales.

Estas recomendaciones son aplicables de manera similar en el entorno Prolog. Dividir diferentes predicados en módulos y ordenarlos referencialmente asegura que aquellos predicados con funciones parecidas estén próximos entre sí. En cuanto al tamaño de las líneas, se recomienda una longitud de menos de 60 caracteres, con un límite máximo de 80 caracteres; la herramienta final desarrollada tiene un promedio de 38 caracteres por línea. Respecto a la recomendación de indentación, se adopta una estructura uniforme para todas las reglas, donde cada sentencia está indentada en una nueva línea. Si los argumentos de alguna sentencia o el encabezado de la regla superan el límite de 80 caracteres por línea, los argumentos se alinean con una indentación mayor que la sentencia y se colocan en líneas separadas.

Aplicando estas recomendaciones al predicado `receta_con_carne/1` descrito en el apartado 3.1 se obtendría la siguiente declaración:

```
receta_con_carne(X) :-
    contiene(X,Y),
    carne(Y).
```

5.2.3. Código

Las recomendaciones de código se centran en la lógica del programa, en cómo se organizan las funciones y en las líneas de código que las componen. Algunas de las principales recomendaciones relacionadas con esta categoría son:

- Mantener la consistencia. Un formato uniforme en la escritura del código facilita su lectura y permite hacer suposiciones que agilizan su comprensión.
- Las funciones deben tener un único propósito. Representan un paso en el razonamiento del programa; definir un solo propósito clarifica su función y reduce la posibilidad de errores de dependencia.
- Encapsular condiciones de límite. Hacerlo aclara el razonamiento detrás de un límite sin necesidad de añadir un comentario, y disminuye la posibilidad de cometer errores del tipo *Off-by-One*, donde se omiten elementos durante el procesamiento debido a una condición de parada incorrecta en un bucle.
- Preferir menos argumentos por función. Tener demasiados argumentos puede complicar la lectura y comprensión de una función. Una gran cantidad de argumentos podría ser indicativo de una inadecuada encapsulación de objetos o estructuras de datos.

En el entorno de Prolog, varias de estas recomendaciones se pueden aplicar directamente, como mantener la consistencia entre los diferentes predicados y sus contenidos. En lo que respecta a las recomendaciones sobre funciones, éstas pueden adaptarse a predicados. Es esencial que cada predicado tenga un hecho bien definido y no implique una búsqueda más extensa de lo que determina, de manera análoga a las funciones en un lenguaje imperativo. De forma similar, la idea de minimizar los argumentos en una función también puede aplicarse a los argumentos de un predicado. Sin embargo, es común que en Prolog haya un mayor número de argumentos por predicado que en una función de un lenguaje imperativo debido al carácter recursivo de estos.

El concepto de "bucle" no existe en Prolog, puesto que es una estructura típicamente condicional que dicta cómo se ejecutan las instrucciones, no la lógica subyacente. Para definir una estructura con múltiples repeticiones en Prolog, se hace referencia a sí misma de forma recursiva, estableciendo una condición de terminación en vez de una cuenta de índice. Por ello, no es necesario encapsular las condiciones de límite; en un contexto declarativo, estas siempre se expresan de forma explícita, proporcionando así la información adicional requerida.

5.2.4. Nombres y comentarios

La categoría de "nombres y comentarios" se refiere a la legibilidad de los elementos en el código, considerando los nombres de variables, funciones y los comentarios que se incluyan en el programa. Algunas recomendaciones para esta categoría son:

- Elegir nombres descriptivos y no ambiguos. Los nombres completos, sin abreviaturas, que describen claramente lo que representan, facilitan la comprensión sobre la función del elemento en cuestión y hacen más intuitivo el código.
- Realizar distinciones con significado. Elegir nombres distintos para conceptos similares, que destaquen sus diferencias, ayuda a clarificar y a reducir el riesgo de usarlos de manera incorrecta.
- Evitar las explicaciones de comentarios, el código debería ser autoexplicativo. Reducir la cantidad de comentarios mediante nombres descriptivos facilita la lectura del código y permite presentar la lógica de manera más clara.
- Evitar redundancias en los comentarios. Los comentarios repetitivos pueden hacer más lenta la lectura del código y generar incoherencias si este se modifica durante el desarrollo.

Todas las recomendaciones mencionadas se han implementado en el desarrollo del programa, ya que son útiles en cualquier lenguaje de programación. En el programa Prolog, los comentarios se usan exclusivamente para describir estructuras que alteran la búsqueda de soluciones, pues estas se desvían del paradigma declarativo y son una fuente común de errores. Dado que la función de la aplicación es actuar como herramienta de instrumentación, esta debe manejar estructuras de datos complejas en un lenguaje poco convencional, describiendo los procesos internos de un intérprete Prolog. Por ello, se presta especial atención al nombramiento de los términos en cada predicado y se mantiene consistencia entre ellos para simplificar la lectura del programa.

5.2.5. Excepciones a la norma

Es importante destacar que los principios y recomendaciones establecidos para el desarrollo de esta herramienta no son inmutables y pueden adaptarse según el caso par-

ticular. En determinadas situaciones, podría ser conveniente apartarse de los principios establecidos con el objetivo de obtener un código más legible y resiliente.

La función `names_to_launchers/3` acepta como primer argumento una lista de predicados y, como segundo, una lista de predicados modificados cuya funcionalidad ha sido alterada. Esto forma parte del diseño del proceso de instrumentación, donde se crean predicados para medir posteriormente la cobertura durante una ejecución.

Para ilustrar las modificaciones hechas sobre los principios de código limpio, se describe un predicado simplificado llamado `predicado_a_lanzadera/2` con un funcionamiento similar.

```
predicado_a_lanzadera(Cabeza :- Cuerpo, Lanzadera) :-
    calculos_iniciales,
    Lanzadera = (Cabeza :- anotar_cobertura, Cuerpo).
```

Este predicado contiene tres símbolos `:-`, aunque comúnmente sólo aparece uno por regla. A pesar de su simplificación, puede resultar difícil seguir el razonamiento detrás de lo que realiza la función. Al modificar la estructura y separación, sin seguir las recomendaciones habituales, el predicado se puede reescribir de la siguiente manera:

```
predicado_a_lanzadera(Cabeza :- Cuerpo, Lanzadera) :-
    calculos_iniciales,
    Lanzadera = (
        Cabeza :-
            anotar_cobertura,
            Cuerpo).
```

Esta adaptación facilita la lectura del código, y para este predicado en particular, ofrece una solución más adecuada que la proporcionada por las reglas originales. Durante el desarrollo de la herramienta, se permite la adopción de ajustes similares en aquellos casos donde se considera que ofrecerán un código más limpio que la opción predeterminada.

5.3 Tecnología utilizada

El análisis de las tecnologías utilizadas en la solución, el ámbito tecnológico y las diferentes alternativas propuestas se ha llevado a cabo de manera incremental a medida que se introducían como opciones viables para la solución propuesta.

En el apartado 4.2 se han evaluado las distintas tecnologías aplicables al problema. Se ha considerado el uso de Prolog como herramienta principal para el desarrollo debido a su portabilidad, facilidad de uso, valor académico y compatibilidad con herramientas existentes.

Dentro de los intérpretes de Prolog, se ha elegido SWI-Prolog por su popularidad y funcionalidades adicionales que respaldan el programa. Sin embargo, se desarrolló la solución teniendo en cuenta que debería ser fácilmente adaptable a otras versiones de Prolog.

El conocimiento personal de las herramientas antes de iniciar el desarrollo era nulo así que, antes de considerar el trabajo más allá de ser un programa relacionado con el lenguaje Prolog, se destinaron aproximadamente 100 horas al aprendizaje de los aspectos fundamentales del lenguaje para adquirir habilidad en la escritura de programas.

Algunas soluciones evaluadas, aunque se consideraron poco viables, se implementaron con el propósito de explorar en profundidad la funcionalidad de la herramienta

y obtener una comprensión más detallada de los requisitos. En total, estas soluciones requirieron aproximadamente 70 horas de desarrollo.

CAPÍTULO 6
Desarrollo de la solución propuesta

6.1 Desarrollo

Esta Sección estará basada en los conceptos explorados en el apartado 4.4 para contrastar el plan de trabajo inicialmente propuesto con el desarrollo final de la aplicación.

El plan de trabajo dividía las fases de desarrollo en cuatro *Sprints*, cada uno con objetivos específicos. A lo largo del desarrollo, estos objetivos se mantuvieron constantes y se lograron cumplir puntualmente en cada *Sprint*. Por ello, se puede afirmar que el desarrollo se ajustó al plan general propuesto y resultó ser exitoso.

Sin embargo, algunos de los objetivos generales, y consecuentemente los incrementos propuestos para el *Sprint* al que pertenecían, no tuvieron el impacto esperado en el desarrollo final, mientras que otras áreas del desarrollo ganaron protagonismo.

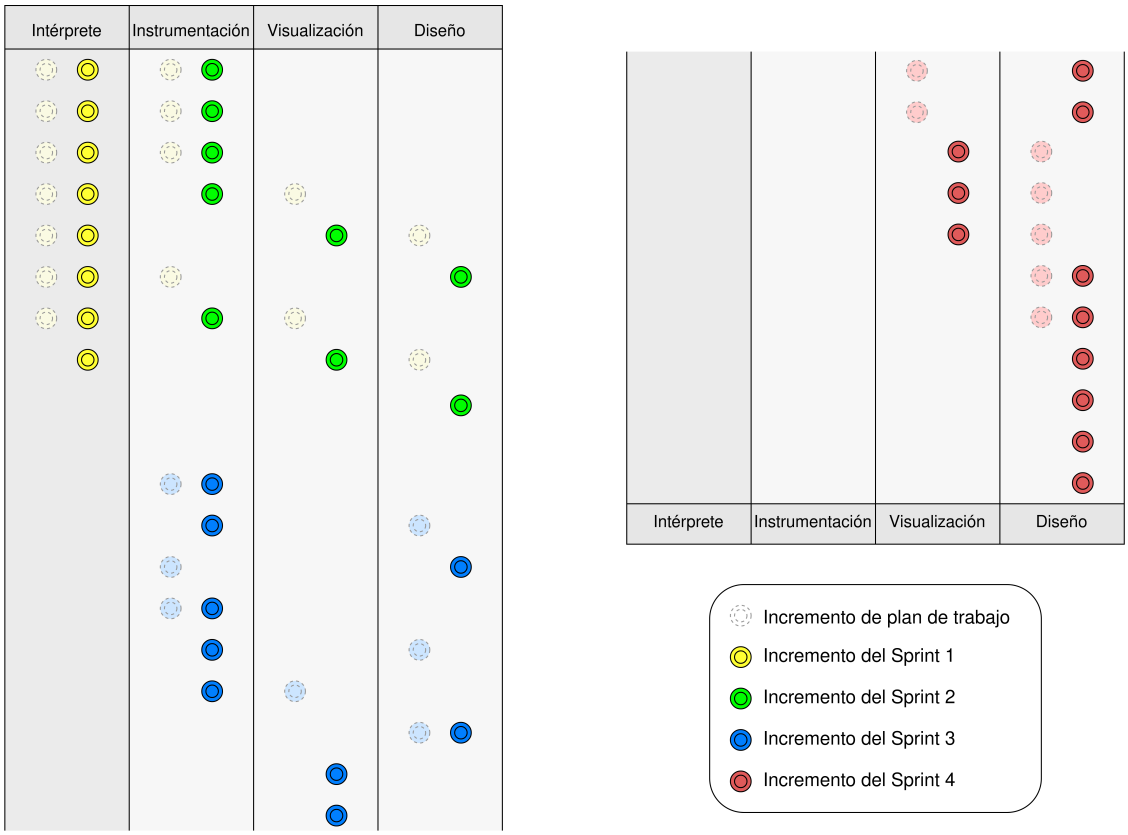

Figura 6.1: Distribución de Incrementos en las partes del programa

En la Figura 6.1 se muestra gráficamente los incrementos establecidos en el plan de trabajo para cada *Sprint*, así como los resultados finales de estos. Cada incremento se representa secuencialmente con dos círculos concéntricos del color asignado al respectivo *Sprint*. Los círculos de línea punteada y color claro indican los incrementos previstos en el plan de trabajo para cada *Sprint*. Dichos incrementos se dividen en cuatro secciones del programa:

1. Intérprete: Se refiere a la prueba de concepto ejecutada en el primer *Sprint* que muestra una cobertura simplificada. Esta Sección del programa, al igual que el *Sprint* en el que se desarrolla, es funcionalmente independiente del prototipo final.
2. Instrumentación: Abarca todas las partes de la herramienta involucradas en el proceso de instrumentación, ejecución y recopilación de datos de cobertura del programa.
3. Visualización: Los incrementos en esta Sección están directamente vinculados a la interfaz del programa, presentando los datos una vez que se ha configurado y se ha llevado a cabo la cobertura.
4. Diseño: Esta Sección incluye las funciones del programa que no pertenecen ni a la instrumentación ni a la visualización. Puede abarcar actividades como la refactorización de componentes, pruebas al sistema, distintas opciones y modos de ejecución del programa y el desarrollo de documentación, entre otros.

6.1.1. Realización de los *Sprints*

Sprint 1 - Prueba de concepto

Este *Sprint* tenía una dirección diferente a los demás, ya que su propósito era desarrollar una prueba de concepto utilizando un intérprete ya existente, con el fin de presentar resultados iniciales que permitieran evaluar la viabilidad del producto.

La mayoría de las actividades de este *Sprint*, desde el Incremento I1 hasta el I3, se planteaban con un enfoque exploratorio. A través de ellos se pretendía observar la implementación de las herramientas existentes no solo para realizar las modificaciones adecuadas, sino también con un propósito didáctico, explorando diferentes herramientas para construir una base que capitalizara sus fortalezas.

Durante el desarrollo del *Sprint*, la única modificación en los Incrementos fue la inclusión de uno adicional entre I3 e I4: "Seguimiento y control de una traza de ejecución de `'prologt'` junto con `'tokenize'`".

Sprint 2 - Instrumentador básico

Este *Sprint* señala el inicio del desarrollo de la herramienta final. Debido a ello, se espera que sus Incrementos y objetivos no difieran significativamente de lo establecido en el plan de trabajo. La meta de esta iteración es alcanzar una solución mínima viable, es decir, producir un prototipo con las funciones esenciales sobre el cual se trabajará en los *Sprints* posteriores.

Durante el desarrollo de este *Sprint*, se lograron los objetivos, añadiendo dos incrementos:

- Incremento entre I3 y I4: "Implementación de un sistema de registro de cláusulas sin cobertura para mostrarlas en los resultados."

- Incremento entre I6 y I7: "Ampliación de la herramienta de lectura de predicados para incluir sentencias y Gramáticas de Cláusulas Definidas."

Estos incrementos se incluyeron debido a características de la implementación que no se habían contemplado en el diseño inicial. Estas desviaciones en la planificación son comunes, y la metodología Scrum está diseñada para adaptarse y resolverlas, introduciendo las correcciones necesarias al plan.

Sprint 3 - Instrumentador modular

En esta iteración, se propone desarrollar una extensión del prototipo mínimo del *Sprint* anterior, ampliando sus funcionalidades y asignando recursos para reestructurar los predicados, facilitando así futuras modificaciones.

Un cambio significativo que se introduce en el desarrollo de este *Sprint* es la función de lanzadera (detallada en el apartado 6.2.1). Este cambio surge debido a una funcionalidad del programa que aportaría valor al mismo y que se identifica durante el proceso de desarrollo. Al considerar esta función en el *Sprint*, es necesario evaluarla en función de su valor añadido y del tiempo adicional que requerirá. La decisión de incluirla en la implementación se tomó en la etapa de Revisión del *Sprint* anterior entre el tutor y el alumno. En la Planificación de este *Sprint*, se decidió incorporarla como el primer incremento.

Para esta iteración se añadieron dos incrementos:

- Incremento antes de I1: Implementación de lanzaderas de predicado para integrar la medida de cobertura directamente en una consulta.
- Incremento entre I2 y I3: Implementación de consultas secuenciales basadas en una única medida de cobertura.

Sprint 4 - Interfaz

El *Sprint* 4 persigue dos objetivos principales. En primer lugar, se centra en la interfaz de la aplicación, mejorando la visualización de la cobertura y expandiendo su funcionalidad a diferentes parámetros y entornos. Por otro lado, este *Sprint* incluye Incrementos para finalizar el desarrollo de la herramienta, añadiendo detalles finales y acabando tareas pendientes de secciones anteriores.

Dada su naturaleza como último *Sprint* y la posibilidad de haber realizado cambios en *Sprints* previos, no es sorprendente que exista una variación significativa en los incrementos de este *Sprint*. Como resultado, este requiere 11 incrementos en lugar de los 7 inicialmente propuestos.

Los incrementos añadidos son:

- Incremento antes de I1: Reorganización de la lectura de predicados del fichero, traspasando el procesamiento a un único predicado para mejorar el mantenimiento.
- Incremento antes de I1: Reestructuración del flujo de datos entre los predicados de instrumentación, incorporando numeración a las cláusulas e información de su tipo.
- Incremento entre I1 y I2: Creación de un sistema de visualización modular que presenta los resultados de manera adaptable y fácil de modificar.

- Incremento entre I1 y I2: Reorganización del sistema de registro y procesamiento de cobertura para el módulo de visualización.
- Incremento entre I5 y I6: Pruebas del sistema.

Los incrementos previos a I1 y el Incremento entre I5 e I6 se introducen debido a cambios en *Sprints* anteriores. Estos cambios hacen necesario refactorizar el código para añadir la funcionalidad pendiente o mejorar la implementación actual. Al igual que en el *Sprint* anterior, estas modificaciones surgieron de la etapa de Revisión del *Sprint* previo, donde se evaluaron los resultados del *Sprint* en relación con los objetivos establecidos.

Los incrementos entre I1 e I2 surgen a raíz de un replanteamiento en la generación de la tabla de resultados. Estos cambios mejoran la propuesta inicial presentada en el plan de trabajo y se examinan con detalle en el apartado 6.2.5.

Además, en este *Sprint* se elimina uno de los Incrementos planteados en la planificación, ya que este deja de tener sentido con los cambios introducidos al sistema de visualización:

- I1: Desarrollo del módulo de visualización para permitir la configuración de la tabla mostrada.

6.2 Elementos del diseño

En este apartado se describen algunas partes de la implementación de especial interés, debido a su relevancia en el sistema, su utilización de los principios de diseño establecido, su complejidad o su criticidad en la ejecución.

6.2.1. Resultados integrados en la consola

Una de los puntos clave que se recalca en los casos de uso es la rapidez de uso de la herramienta. Se refuerza que la herramienta debe poder activarse de manera rápida sin requerir mucha configuración para facilitar su uso en un entorno de desarrollo y más allá de una medida únicamente utilizada en la fase de pruebas.

En la fase de desarrollo la persona que lleva a cabo el desarrollo tendrá un fichero con un programa, por ejemplo, `'base_conocimientos.pl'` y estará realizando consultas sobre este para validar el funcionamiento de los predicados y cláusulas. Tal y como se detalla en la Sección 4.1.4 (Caso de uso CU01), si la persona usuaria necesita medir la cobertura de una consulta sobre la base de conocimientos, deberá realizar una llamada a la herramienta en la consola en la que especifique los parámetros de la ejecución, así como el nombre del fichero, `'base_conocimientos.pl'`, y la consulta que desea medir. Puesto que la herramienta es accesible desde la propia consola de Prolog en la que se realiza el desarrollo, en la versión más reducida será necesario únicamente realizar una consulta con la forma:

```
?- cobertura('base_conocimientos.pl', consulta(Argumento)).
```

En las primeras etapas de desarrollo y en los casos de uso, este formato de consulta estaba previsto para lograr el objetivo de rapidez. Sin embargo, durante el proceso de desarrollo, se encontró una alternativa mejor que utiliza la instrumentación para facilitar el lanzamiento de consultas, haciendo el uso de la herramienta más transparente. Con esta alternativa integrada en las consultas se puede realizar la instrumentación de un fichero

con un predicado, a partir del cual cualquier consulta sobre los predicados del fichero mostrará, además de su resultado, la cobertura de código. Es decir, con la consulta:

```
?- cargar_cobertura('base_conocimientos.pl').
```

Se realizará la instrumentación de `'base_conocimientos.pl'` de tal manera que al lanzar la consulta:

```
?- consulta(Argumento).
```

Se mostrará una tabla en la consola con los resultados de la cobertura, además del resultado de la consulta, haya unificado o no.

Para implementar esta función es necesario hacer uso de varios predicados de instrumentación y modificaciones a la búsqueda de solución. Partiendo del ejemplo anterior, el fichero `'base_conocimientos.pl'` puede tener una definición del predicado `consulta/1` de por ejemplo:

```
consulta(0) :- write('primera clausula').
consulta(1) :- write('segunda clausula').
```

Si el argumento de la consulta es `0`, este se unificará con la primera cláusula y se imprimirá el mensaje "primera clausula" en pantalla. Si el argumento es `1`, se unificará con la segunda cláusula y mostrará el mensaje "segunda clausula".

En la instrumentación, `consulta/1` será renombrado a `consulta_oculto/1` para que su ejecución sea transparente.

```
consulta_oculto(0) :- write('primera clausula').
consulta_oculto(1) :- write('segunda clausula').
```

El predicado `consulta/1` se usará para calcular la cobertura y devolver los resultados. Por lo tanto, deberá:

- Detectar la primera vez que se realiza la consulta al predicado, mostrando la cobertura solo una vez.
- Indicar las unificaciones que el predicado realizaría de la misma manera que si no estuviera instrumentado.
- Si la consulta al predicado falla, el predicado instrumentado deberá fallar de igual manera, pero mostrando previamente la cobertura.
- Si el predicado se consulta desde la base de conocimientos durante la consulta de otro predicado, por ejemplo, a través de una regla o el predicado `call`, este no deberá mostrar la cobertura.

Para detectar si la consulta es la primera vez que se accede al predicado y garantizar que la cobertura se muestre solo una vez, se emplea el hecho `primera_consulta_realizada/0`. Este hecho se añade dinámicamente a la base de conocimientos en la primera llamada a `consulta/1` y se elimina solo cuando ha concluido la llamada inicial, ya sea que haya sido exitosa o no haya podido unificar.

Para garantizar que se muestre la cobertura y que el hecho `primera_consulta_realizada/0` se elimine de la base de conocimientos incluso si la consulta falla, se usan cortes rojos. Estos modifican la búsqueda de la solución y guían al intérprete hacia una cláusula diferente del predicado en caso de fallo.

Esta funcionalidad se puede reducir a tres cláusulas, que representarán 3 posibilidades en la consulta del predicado:

1. *No se ha realizado la primera consulta (`primera_consulta_realizada/0` no está en la base de conocimientos).* En este caso se deberá añadir `primera_consulta_realizada/0` a la base de conocimientos, consultar el predicado de `consulta_oculto/1` y medir la cobertura.

2. *La primera consulta ya se ha realizado (`primera_consulta_realizada/0` está en la base de conocimientos).* Aquí, se debe consultar el predicado `consulta_oculto/1` y continuar con la búsqueda.

3. *La primera cláusula ha fallado porque `consulta_oculto/1` no ha unificado con la base de conocimientos.* En este escenario, es necesario registrar la cobertura y producir un fallo "artificialmente" para mostrar el resultado adecuado de la consulta a `consulta_oculto/1`.

A continuación se muestra una versión simplificada de los predicados que se añadirían a la instrumentación del fichero `'base_de_conocimientos.pl'` para implementar esta funcionalidad en `consulta/1`. En la herramienta, estas tres cláusulas que se incorporan a todos los predicados se denominan "predicados lanzadera".

```
% Ya se ha realizado la primera consulta
consulta(Argumento) :-
    primera_consulta_realizada,
    !,
    consulta_oculto(Argumento).

% No se ha realizado la primera consulta
consulta(Argumento) :-
    assert(primera_consulta_realizada),
    consulta_oculto(Argumento),
    !,
    mostrar_cobertura.

% La primera cláusula ha fallado
consulta(Argumento) :-
    mostrar_cobertura,
    fail.
```

6.2.2. Lanzaderas

El sistema que permite integrar la medición de cobertura directamente en la consola mediante predicados "lanzadera" facilita la realización de medidas sobre la cobertura antes de la unificación con las cláusulas específicas de cada predicado. En el ejemplo anterior, si añadimos un predicado para medir la cobertura de `consulta_oculto/1`:

```
consulta_oculto(0) :- medir_cobertura, write('primera clausula').
consulta_oculto(1) :- medir_cobertura, write('segunda clausula').
```

Esto solo tendrá en cuenta las consultas que unifican con las cláusulas. Por lo tanto, una consulta que no unifique con ninguna cláusula del predicado no registrará ninguna medida:

```
?- consulta_oculto(2).
```

Sin embargo, la lanzadera mencionada anteriormente unifica los argumentos del predicado con una variable, por lo que no se realiza ninguna instanciación. Esto permite medir la cobertura de las consultas al predicado antes de la unificación de sus argumentos. Con una modificación en la primera cláusula de la lanzadera para incorporar una sentencia de cobertura, se pueden tomar medidas y detectar si las variables están instanciadas o si se han creado puntos de decisión:

```
% Ya se ha realizado la primera consulta
consulta(Argumento) :-
    primera_consulta_realizada,
    !,

    % Medidas de cobertura en la lanzadera
    medir_instanciados(Argumento),
    puntos_de_decision(Argumento),

    consulta_oculto(Argumento).

% No se ha realizado la primera consulta
(...)
```

6.2.3. Módulo utils

El módulo "utils" contiene predicados generales que son empleados por varios módulos del programa. Estos predicados son notables por su uso recurrente en distintas etapas de la ejecución.

El predicado `utils:rearrange/4` acepta los siguientes argumentos:

1. Una lista de entrada que será iterada para generar un resultado.
2. Un término que se unifica individualmente con cada elemento de la lista de entrada.
3. Un término que se unifica individualmente con cada elemento de la lista de salida, y cuyas variables se instancian en concordancia con el primer término.
4. Una lista de salida que contiene los términos del tercer argumento, después de haber unificado el segundo argumento con el elemento correspondiente de la primera lista.

A modo de ejemplo, este predicado puede usarse para obtener los valores de un diccionario representado como una lista de `'<Clave>- <Valor>'`:

```
?- utils:rearrange(Diccionario, Clave - Valor, Valor, Valores).
```

Como se observa, incluso sin conocer la implementación de `utils:rearrange/4`, su diseño es sencillo, intuitivo y aplicable a diversos escenarios.

Otra característica de la unificación en Prolog que puede ser una herramienta valiosa en el diseño de código es la capacidad de usar los argumentos de un predicado tanto para entrada como para salida de información en el flujo del programa.

6.2.4. Documentación

Como parte del diseño de la herramienta, se ha incluido una estructura sistemática y organizada de comentarios para explicar el funcionamiento de cada predicado y módulo del programa.

La documentación del programa se ha redactado siguiendo las directivas de la biblioteca PlDoc [1]. Esta establece un formato para comentarios estructurados dentro del propio código fuente que permite generar archivos de documentación de manera automática e integrada en Prolog, ya sea como archivos LaTeX o como una serie de archivos HTML.

Estos comentarios estructurados siguen un formato similar al de JavaDoc [2]. Antes de cada predicado, se introduce un comentario con las siguientes secciones:

1. El nombre del predicado junto con sus argumentos y tipos.
2. Una breve descripción que indica la función del predicado.
3. Una lista de los parámetros de entrada para el predicado, acompañados de una breve descripción.

A diferencia de JavaDoc, utilizado en el lenguaje Java, los argumentos de un predicado en Prolog no tienen un tipo explícito ni una instanciación clara. Además, el predicado puede funcionar de manera determinista o no determinista, dependiendo de los puntos de elección que establezca. Las especificaciones de PlDoc sugieren que estas características se detallen ya que forman parte del diseño del predicado y delimitan su uso previsto, informando a la persona programadora sobre qué formato puede anticipar un comportamiento adecuado.

Por ejemplo, los comentarios de documentación para la función `utils:rearrange/4`, especificada en la Sección anterior, son:

```
rearrange(?Original : list, +Mask_Original : term,
                +Mask_Rearranged : term, ?Rearranged : list) is det.

Succeeds after matching <Original> element-wise with <Mask_Original>,
giving the bounded <Mask_Rearranged> values for the corresponding element in
<Rearranged>.

For every element, <Mask_Original> and <Mask_Rearranged> are copied as fresh
variables retaining the bindings between each other. This way each
element in <Original> is matched with a new <Mask_Original> term to
generate a <Mask_Rearranged> term.

@param Original         The list of elements matching with <Mask_Original>.
@param Mask_Original    The term matching with each element in
                        <Original> and bounded to <Mask_Rearranged>.
@param Mask_Rearranged  The term bounded to <Mask_Original>.
@param Rearranged       The list of each bounded <Mask_Rearranged>.
```

Para el caso de `utils:rearrange/4` el predicado es determinista, ya que no genera puntos de decisión y, por lo tanto, proporciona una única unificación bajo cualquier combinación de argumentos, siempre que estos respeten los patrones de instanciación.

[1] `https://www.swi-prolog.org/PlDoc.txt)`

[2] `https://docs.oracle.com/javase/8/docs/technotes/tools/windows/javadoc.html`

Los patrones de instanciación de los argumentos se señalan con un símbolo previo al nombre del argumento. En este predicado, el primer y el último argumento llevan el símbolo **'?'**, mientras que el segundo y tercero llevan el símbolo **'+'**.

Para las listas en el primer y último argumento, el símbolo **'?'** señala que pueden estar instanciados, parcialmente instanciados o no instanciados al momento de invocar el predicado. Esto significa que es posible pasar variables no instanciadas para estos argumentos, listas con algunos elementos no instanciados o listas totalmente instanciadas, y el comportamiento no se desviará del descrito.

En cuanto a los términos del segundo y tercer argumento, el símbolo **'+'** indica que deben estar parcial o completamente instanciados. Es decir, el término puede contener variables, siempre que su estructura concuerde con su tipo.

6.2.5. Generación de la tabla

El módulo `visualizer`, encargado de mostrar los resultados de la cobertura en la terminal, es de especial importancia ya que representa la interfaz a través de la cual se comunican los resultados de la cobertura a la persona usuaria. Además, es una Sección del programa donde se prevé una alta probabilidad de mantenimiento debido a posibles modificaciones. Cualquier cambio en la medición de la cobertura o en las variables que se evalúan durante la ejecución puede resultar en variaciones en la presentación de los datos.

Debido a esto, la tabla se genera de manera modular sobre una estructura básica, organizando los datos en distintos módulos que se integran en la tabla de formas específicas. El propósito de este diseño es adaptar la visualización de resultados a posibles cambios que puedan surgir en el desarrollo futuro de la herramienta:

1. *Más medidas de cobertura.* Más allá de los 3 parámetros de cobertura que se evalúan en la herramienta actual, podría surgir la necesidad de medir parámetros adicionales.

2. *Menos medidas de cobertura.* Al igual que en el punto anterior, una extensión útil podría ser la implementación de medidas de cobertura más generales que las actuales. Por ejemplo, se podrían medir los accesos a un módulo o la cobertura general de cada predicado.

3. *Orientación de la tabla.* La orientación actual de la tabla, evaluada en el apartado 4.2.5, es considerada la mejor alternativa para los casos de uso de la herramienta. Sin embargo, otra disposición de los datos podría ser más adecuada para una aplicación particular.

La información obtenida de la cobertura se representa mediante dos estructuras: listas y clasificaciones. Las listas representan una secuencia de elementos pertenecientes a una categoría de la cobertura o un elemento de una lista, mientras que la clasificación denota una subdivisión de elementos que pertenecen a una lista o categoría.

Por ejemplo, los predicados de un programa pueden representarse como una lista, dado que simbolizan una secuencia de elementos dentro de una categoría. Sin embargo, si junto a esta lista de predicados se muestra la cobertura de cada uno, entonces la categoría que diferencia al predicado de su cobertura adopta la estructura de una clasificación.

Esta diferenciación estructural permite representar datos de diversas maneras: la lista se desarrolla verticalmente y la clasificación, horizontalmente, para componer una tabla.

Además, cada categoría alberga una serie de valores que indican sus dimensiones. Basándose en estos valores, se incorporan espacios y se distribuyen las tabulaciones en la tabla resultante.

En la Figura 6.2 se ilustran las estructuras verticales y horizontales, así como sus valores respectivos. Los rectángulos azules contienen un valor específico de la tabla de cobertura (esto podría ser el nombre de un predicado, el número de una cláusula o la cobertura obtenida) dependiendo del nivel en el que se encuentren. Los rectángulos verdes encapsulan otra estructura, que podría ser vertical u horizontal. Los rectángulos blancos simbolizan el espacio en blanco intercalado entre los datos, ayudando a diferenciar la configuración de la tabla final.

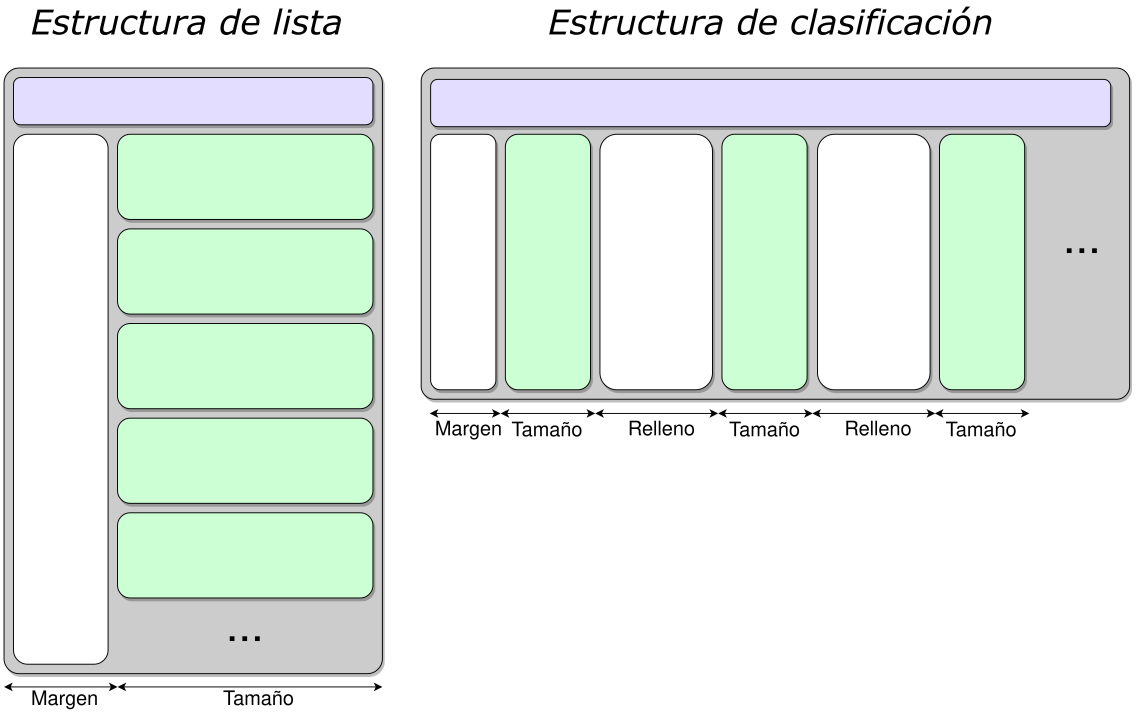

Figura 6.2: Representación de las estructuras de la tabla

Tras diseñar la estructura de la tabla, definir los parámetros de cada nivel y distribuir las medidas según su posición, esta se traduce en una serie de instrucciones de formato que se introducirán como argumento al predicado `format/2`. El resultado es una tabla modular adaptada a las especificaciones de las medidas individuales.

6.3 Mejoras

En esta Sección se proponen mejoras para diversas características del diseño de la aplicación. La metodología Scrum elegida para el desarrollo contempla un diseño iterativo, en el que se llevan a cabo evaluaciones constantes del trabajo y se desarrollan soluciones incrementales hacia un objetivo definido. Dicho objetivo puede sufrir modificaciones a lo largo del proceso. En un proceso donde la solución se mejora de manera incremental, se debe establecer un punto funcional y temporal en el que esta se considere finalizada. A partir de entonces, las propuestas para perfeccionarla se considerarán como mejoras para el futuro. Estas mejoras están contempladas en este proceso, pero no se han aplicado debido al límite temporal y de complejidad del proyecto. La implementación, desde el aprendizaje de los entornos hasta el prototipo final, se ha limitado a las 300 horas de desarrollo establecidas en el plan de trabajo.

6.3.1. Estructuras de datos

El lenguaje Prolog no ofrece estructuras de datos más complejas que una lista. Observado desde una perspectiva puramente declarativa, la lista en Prolog se compone de múltiples términos complejos con dos argumentos: el primero contiene un elemento de la lista, y el segundo contiene el resto de la lista.

```
?- [1,2,3] = '.'(1, '.'(2, '.'(3, []))).
true.
```

A pesar de tener acceso a estructuras limitadas, las capacidades que ofrece la unificación como herramienta para descomponer las distintas partes de un término permiten un acceso transparente y cumplen con las recomendaciones de código limpio.

En el módulo `'logger'`, donde se añaden las sentencias de lanzamiento para los predicados que se quieren medir, es necesario almacenar información sobre estos. Para generar los predicados de lanzamiento, el módulo utiliza una lista con los identificadores del predicado en la forma `Nombre/Aridad`, una lista con sus tipos (sentencia/regla/dcg) y una lista que contiene el contenido de las cláusulas en la forma `Cabeza - Cuerpo`.

En los módulos de `'logger'` y `'utils'` se desarrollan distintos predicados (como `rearrange/4`, cuyo funcionamiento se describe en el apartado 6.2.3) que permiten extraer los datos de estas estructuras de manera sistemática y directa. Sin embargo, un diseño óptimo agruparía estas estructuras relacionadas en una única estructura de datos cuyo orden sea transparente para el módulo `'logger'`.

De manera similar al tratamiento interno que Prolog da a las listas, esta estructura podría crearse en un módulo separado y su acceso podría limitarse a un predicado declarado dentro del programa. Por ejemplo, el acceso a los tipos de los distintos elementos podría cambiar de una unificación explícita en el predicado:

```
escribir_tipos(Datos) :-
    Datos = [Tipos | _],
    write(Tipos).
```

A una llamada a un predicado externo:

```
escribir_tipos(Datos) :-
    base_conoc:contiene(Datos, tipos, Tipos),
    write(Tipos).
```

En este ejemplo, el predicado `contiene/3` del módulo `'base_conoc'` extrae los tipos de los predicados de una estructura transparente a los módulos exteriores y contenida en la variable `Datos`.

6.3.2. Tabla de cobertura

Para generar la tabla que se muestra al solicitar la cobertura, se efectúan varios recorridos a través de la estructura que contiene los datos de dicha cobertura. Estos recorridos cumplen con las siguientes funciones:

1. Obtener la estructura de la tabla, estableciendo los datos en formato vertical y horizontal (listas y categorías).

2. Determinar las dimensiones de la tabla, ajustando los espacios para prevenir la superposición de las diferentes columnas.

3. Procesar los datos y las dimensiones para obtener una lista con el formato y los datos de cada fila de la tabla.

4. Convertir los formatos de fila en una secuencia de caracteres y términos para ser introducidos en el predicado `format/2`.

Esta estructuración permite visualizar los datos generados en cada etapa del proceso de manera secuencial. Dicha visibilidad facilita el proceso de corrección de errores y posibilita el desarrollo de la tabla de forma incremental. La creación de la tabla, debido a la complejidad de la estructura de datos que genera, se beneficia de este enfoque incremental.

A pesar de estas ventajas, la naturaleza secuencial del procedimiento altera la perspectiva del código declarativo, requiriendo intervenciones en las listas que contienen los datos. Además, iterar sobre las listas repetidas veces disminuye la eficiencia del código y almacena valores para su uso posterior, los cuales podrían obtenerse mediante unificación.

Una mejora al código actual implicaría ocultar la complejidad del manejo de las estructuras en un módulo a parte, responsable de administrar una estructura de datos que guarde toda la información. Esta mejora se podría implementar de manera similar a lo propuesto en el apartado anterior.

De esta manera, al eliminar parte de la complejidad, la generación de la tabla podría llevarse a cabo en un único recorrido de los datos de cobertura, realizando unificaciones con valores indeterminados para resolverlos durante el retorno de la llamada recursiva.

6.4 Uso de la herramienta

Para implantar la herramienta en el sistema destino, basta con descargar los ficheros Prolog que contienen el código fuente de la aplicación. Estos están disponibles a través de un enlace en la plataforma *GitHub* [3].

Una vez descargado el programa, es necesario tener SWI-Prolog instalado en el sistema para poder ejecutarlo. La herramienta se ejecuta directamente desde la línea de comandos. Utilizando una terminal en *bash*, el comando sería:

```
$ ./plcov.pl 'base_conocimientos.pl' -g 'consulta'
```

Para conocer la cobertura de un fichero de pruebas en formato '.plt', que contiene una lista de consultas, se debe usar el comando:

```
$ ./plcov.pl 'base_conocimientos.pl' -f 'pruebas.plt'
```

Para conocer todas las funciones que ofrece la herramienta, se dispone también de una opción de ayuda que muestra las alternativas de ejecución y cómo usarlas. Para acceder a esta lista, se introduce el comando:

```
$ ./plcov.pl -h
```

[3] `https://github.com/`

Para acceder al entorno de consola de SWI-Prolog en bash se introduce el comando:

```
$ swipl
```

Si se utiliza la herramienta dentro del propio entorno de SWI-Prolog, las consultas para activar la cobertura serían:

```
?- [plcov].
true.

?- load('base_conocimientos.pl').
true.
```

Una vez cargado el archivo que contiene el programa, cualquier consulta sobre los predicados del fichero proporcionará una medida de cobertura, además del resultado de la unificación de la consulta. Tomando como ejemplo lo presentado en la Sección 3.1, donde se consulta si la paella es una receta con carne, la ejecución mostraría:

```
?- receta_con_carne(paella).
predicate
   ingrediente/1
      clause    coverage
       1         0
   carne/1
      clause    coverage
       1         1
   contiene/2
      clause    coverage
       1         1
       2         1
       3         0
   receta_con_carne/1
      clause    coverage
       1         1
true.
```

Los resultados de la cobertura indican que se ha realizado un acceso a `receta_con_carne/1` (correspondiente al acceso de la consulta inicial). También se ha accedido a las dos cláusulas iniciales del predicado `contiene/2`. Estas dos cláusulas especifican los ingredientes presentes en la paella, mientras que la tercera se refiere al hervido. Finalmente se accede al predicado `carne/1`, donde se confirma que el conejo es un ingrediente que contiene carne.

Para acceder a las opciones de cobertura y poder visualizar la unificación con las cláusulas de cada predicado, así como la instanciación de las variables en las consultas, se debería realizar una consulta al predicado `load/2`. Esta consulta puede hacerse en la misma ejecución anterior, ya que automáticamente borra las mediciones previas:

```
?- load('base_conocimientos.pl', [ground]).
```

Al ejecutar nuevamente el predicado, se obtendrán resultados que muestran tanto la instanciación de variables en las consultas como la cobertura.

```
?- receta_con_carne(paella).
predicate
   ingrediente/1
      clause    coverage        ground                 coverage
       1         0
   carne/1
      clause    coverage        ground                 coverage
       1         1               carne(g)               2
   contiene/2
      clause    coverage        ground                 coverage
       1         1               contiene(g,ng)         1
       2         1
       3         0
   receta_con_carne/1
      clause    coverage        ground                 coverage
       1         1               receta_con_carne(g)    1
true.
```

La columna 'ground' de la tabla muestra el nombre del predicado con sus argumentos reemplazados por el átomo `'g'`, que indica que el argumento estaba instanciado en esa medida, y `'ng'`, que señala que el argumento no estaba instanciado en esa medida.

En este caso, el predicado `receta_con_carne/1` recibe un argumento instanciado, que es el átomo `paella`. Por su parte, el predicado `contiene/2` se consulta con el primer argumento instanciado a `paella`. El segundo argumento se unificará con un ingrediente que, posteriormente, intentará unificarse con `carne/1`. Se consulta el predicado `carne/1` en dos ocasiones, y en ambas, el argumento está instanciado. Tal como se describe, estas consultas son realizadas por `receta_con_carne/1`, que verifica si cada ingrediente proveniente de `contiene/2` se unifica con el predicado `carne/1`.

El segundo argumento de `load/2` también puede modificarse a `[branch]` para mostrar la unificación con las cláusulas de cada predicado:

```
?- receta_con_carne(paella).
predicate
   ingrediente/1
      clause    coverage        branch      coverage
       1         0
   carne/1
      clause    coverage        branch      coverage
       1         1               [1]         1
                                 []          1
   contiene/2
      clause    coverage        branch      coverage
       1         1               [1,2]       1
       2         1
       3         0
   receta_con_carne/1
      clause    coverage        branch      coverage
       1         1               [1]         1
true.
```

En este caso, el predicado `receta_con_carne/1` tiene solo una cláusula y, por lo tanto, unifica con ella durante la consulta. El predicado `contiene/2` unifica tanto con la prime-

ra como con la segunda cláusula, puesto que ambas tienen el átomo `paella` en su primer argumento. La cobertura de esta unificación se indica como 1, porque al unificar con ambas se crea un punto de elección. Esto significa que no se llega a la segunda cláusula mediante una consulta directa, sino debido al fallo de la primera. Finalmente, el predicado `carne/1` se consulta en dos ocasiones: en la primera no unifica con ninguna cláusula (debido a la expresión `carne(arroz)` que no logra unificar), mientras que en la segunda la unificación es exitosa con la única cláusula existente.

La función `load/2` admite también una lista `[branch, ground]` con ambas opciones para mostrar las dos en una sola ejecución. Posteriormente a haber realizado una consulta de `load/1` o `load/2`, el predicado `query/1` permite medir la cobertura de un fichero de pruebas. Para llevar a cabo esta evaluación, el predicado consulta todos los términos del archivo y muestra la cobertura al finalizar.

```
?- query('fichero_de_pruebas.plt').
```

6.5 Pruebas

Las pruebas, como parte del desarrollo, son fundamentales en cualquier software [Peled, 2001]. Estas garantizan que el programa funcione de manera adecuada, validan las especificaciones y la documentación, y permiten comprobar cualquier modificación al sistema, asegurándose de que no afecte la funcionalidad ya existente.

Este trabajo subraya la importancia de las pruebas, especialmente porque el principal caso de uso del software se manifiesta durante la etapa de prueba del código. Por ello, el análisis realizado en este segmento no solo sirve para validar el correcto funcionamiento del programa, sino también para describir un proceso de pruebas específico para Prolog y destacar el uso de herramientas de medición de cobertura en este contexto.

Las pruebas descritas en este apartado se presentan en el contexto de una serie de evaluaciones llevadas a cabo y documentadas en un sistema que integra todos los casos de uso y objetivos. Aunque estas pruebas se diseñaron y aplicaron a lo largo del proceso de desarrollo, el paso final simplemente las agrupa y añade las esenciales para integrar las distintas funcionalidades de las fases del desarrollo.

En el apartado 4.4.3, donde se detallan los incrementos individuales, cada *Sprint* incluye un incremento dedicado a la prueba del código desarrollado. Esta tarea se ha estructurado en varias fases:

1. **Aplicación de pruebas existentes a las nuevas modificaciones.** Esta fase implica la ejecución de pruebas previamente establecidas en *Sprints* anteriores, con el propósito de asegurar que las secciones del programa desarrolladas con anterioridad sigan funcionando adecuadamente. La intención no es establecer un conjunto de pruebas inmutables a lo largo del proceso de desarrollo, sino utilizar las pruebas existentes para identificar áreas del código que han tenido cambios, y así poder diseñar nuevas pruebas que los aborden. Estas pruebas son comúnmente conocidas como pruebas de regresión [Sommerville, 2016].

2. **Identificación de secciones unitarias del nuevo código.** En consonancia con la fase anterior, es necesario identificar aquellas secciones del código recién desarrolladas, bien sea a través de la estructura de los predicados o las diversas funciones que desempeñan, con el fin de ejecutar pruebas unitarias sobre ellas.

3. **Realización de pruebas.** Una vez reconocidas las diferentes secciones del código elaborado y sus funciones, se plantean distintos casos de prueba para evaluar su

funcionamiento. Estos considerarán tanto su funcionamiento habitual como situaciones límite donde podrían aparecer errores.

6.5.1. Pruebas en Prolog

Antes de describir las pruebas que se han llevado a cabo en el código, es esencial establecer el contexto de efectuar pruebas en un entorno declarativo, específicamente en el lenguaje Prolog. Este lenguaje presenta peculiaridades que pueden facilitar el proceso de pruebas, mientras que otras pueden complicarlo.

Diferencias entre lenguajes imperativos y declarativos

Las pruebas desarrolladas para un programa dependen en gran medida de las pautas de diseño que se han establecido para él. Debido a esto, los cambios que el entorno Prolog presenta en comparación con un entorno imperativo, explorados en el apartado 5.2, también influyen en las pruebas a desarrollar.

Una de las consecuencias de usar la unificación como herramienta principal en la búsqueda de soluciones es la redefinición del concepto de argumentos de entrada y salida. En una función imperativa, se establecen argumentos de entrada y un valor de salida que describen claramente la dirección de la lógica. Por ejemplo, en una función imperativa donde se realiza una suma $a + b = c$, existen dos argumentos a y b de entrada, y la función devuelve un valor c como salida. Sin embargo, en una definición declarativa, esta función se establecería con 3 variables que podrían actuar tanto como entrada como salida. Si las variables c y b están instanciadas, la solución se unificará con la variable a y será determinista; mientras que si solo a está definida, el valor de cc dependerá de b, por lo tanto, no será determinista.

Esta característica, en la que las variables referenciadas por los predicados pueden funcionar en ambas direcciones lógicas (como un argumento instanciado o como una variable a unificar), complica la generación de pruebas. Para asegurarse de que el predicado funcione correctamente, será necesario probar en ambas direcciones y combinando los distintos argumentos. Utilizando el ejemplo anterior de una función que representa una suma, se puede ver cómo esta característica puede dificultar el desarrollo. En el caso de la función de suma imperativa, únicamente se necesita realizar una prueba. Por ejemplo:

```
ASSERT suma(1, 2) IS 3
```

En el caso de la función en estilo declarativo, donde cualquiera de los tres argumentos podría no estar instanciado, para garantizar su correcto funcionamiento se necesitan las siguientes pruebas:

```
ASSERT suma(1, 2, c), c IS 3
ASSERT suma(1, b, 3), b IS 2
ASSERT suma(a, 2, 3), a IS 1
```

Estas pruebas abarcan los casos deterministas del predicado, pero no se enfocan en la parte no determinista. Un ejemplo de este comportamiento no determinista podría ser:

```
?- suma(1, B, C).
B = 0, C = 1;
B = 1, C = 2;
```

```
B = 2, C = 3;
...
```

Este comportamiento específico, donde si b y c no están definidos la búsqueda continúa indefinidamente, es típico de un predicado declarativo. Es, de hecho, lo que se esperaría de dicho predicado. Sin embargo, las herramientas proporcionadas por Prolog no son las más adecuadas para probar este tipo de comportamiento.

Prolog tiene predicados diseñados específicamente para gestionar consultas no deterministas, como `findall/3`, que reúne todas las respuestas posibles a una consulta y las unifica en una lista. Siguiendo con el ejemplo, si quisiéramos comprobar que, ante una consulta donde el segundo y tercer argumento no están instanciados y el primero tiene un valor, la segunda variable comienza en 0 y se incrementa en 1 para las primeras 5 soluciones, podríamos intentar usar la siguiente expresión:

```
ASSERT findall(B, suma(1, B, _), Bs), Bs IS [0,1,2,3,4]
```

No obstante, las herramientas estándar de Prolog definidas en el estándar ISO (Sección 8.10) no ofrecen esta funcionalidad directamente. Por lo tanto, la consulta anterior entraría en un bucle indefinido buscando soluciones para el predicado `suma`, incluso si el tamaño de la lista `Bs` ya se hubiera especificado previamente.

Las complejidades introducidas por el código no determinista son difíciles de comprender, ya que presentan numerosos casos límite que pueden generar comportamientos indeseados. En las pruebas realizadas sobre la herramienta para predicados no deterministas, la verificación se limita a la primera solución, confiando en que el análisis del código garantizará un funcionamiento correcto posterior. Para el caso de `suma/3`, se añadirían las siguientes pruebas:

```
ASSERT suma(1, B, C), !, B IS 0, C IS 1
ASSERT suma(A, 1, C), !, A IS 0, C IS 1
ASSERT suma(A, B, 1), !, A IS 0, B IS 1
```

En resumen, el código declarativo de Prolog facilita la creación de programas más intuitivos y permite definir soluciones que reflejan de manera más precisa el problema. Sin embargo, esta claridad expresiva también hace que las pruebas unitarias, y en especial aquellas que se basan en conceptos de caja blanca (donde se controla el flujo del programa y no solo las soluciones que produce), resulten más difíciles de implementar.

Lanzamiento de errores en Prolog

El manejo de errores en Prolog es diferente al que se hace en programación imperativa. En un contexto puramente declarativo, el concepto de error realmente no existe, ya que el propósito del programa es establecer una distinción entre lo que "es" y lo que "no es". En este escenario, una variable que se instancia al primer argumento de una lista en un predicado, cuando dicha lista está vacía, no representa un error. En su lugar, indica simplemente una situación que "no es" y, por ende, proporciona esa información a la lógica del programa.

Sin embargo, existen conceptos en Prolog que necesitan un mecanismo para lanzar errores. Esto se debe a que no todos los aspectos de un programa están basados puramente en definiciones lógicas. Por ejemplo, un archivo que no existe, un valor introducido de un tipo incorrecto o una conexión interrumpida son circunstancias en las que un programa debería lanzar un error. En herramientas específicas, donde se accede a archivos y

se imprimen valores en consola, se interactúa con información externa. Por lo tanto, se utiliza el lanzamiento de errores para señalar situaciones que se desvían de lo esperado.

En relación con los casos de prueba, estos pueden estar diseñados para esperar un error proveniente de un predicado. Por ello, incluirán instrucciones "catch" cuando sea necesario. La decisión de esperar un error o simplemente un `fail` depende del contexto del predicado en cuestión y está documentada para facilitar el mantenimiento en caso de problemas[Covington et al., 2009].

Pruebas en SWI-Prolog

El estándar ISO de Prolog no proporciona un marco para el desarrollo de pruebas unitarias. Por ello, SWI-Prolog ha implementado la librería `plunit`, que ofrece herramientas para la redacción y ejecución de pruebas unitarias.

Las pruebas se ejecutan mediante los predicados `test/1` o `test/2`, y están rodeados por las directivas `begin_tests/1` y `end_tests/1`. Por ejemplo, utilizando el predicado `suma/3` descrito en el apartado anterior, una prueba tendría la forma:

```
test(suma) :-
    suma(1, 2, 3).
```

Al igual que en otros lenguajes, las pruebas pueden contar con varias opciones especificadas. Estas opciones constituyen el segundo argumento del predicado `test/2`, y permiten invocar predicados de inicialización y de limpieza antes y después de la prueba, respectivamente. Asimismo, ofrecen herramientas para determinar los resultados y para llevar a cabo múltiples consultas. A continuación, se presenta un ejemplo de su aplicación con una condición de fallo:

```
test(suma, [fail]) :-
    suma(1,2,0).
```

Para esta herramienta, se han empleado archivos independientes para el desarrollo de las pruebas, dividiéndolos por módulos y organizando las pruebas según su relación con los predicados del archivo en cuestión. A continuación, se detalla el diseño que se ha seguido para la redacción de las pruebas.

6.5.2. Tipos de pruebas utilizadas

De manera similar al desarrollo del código de la aplicación, se analizaron distintos patrones de diseño convencionales para adaptarlos adecuadamente al entorno Prolog.

En este desarrollo, se consideraron distintos ámbitos y niveles con el objetivo de realizar un análisis variado donde las debilidades de cada método se complementan con las fortalezas de otros.

Pruebas unitarias

Las pruebas unitarias representan el nivel más básico de las pruebas. En ellas, se trata el código como una serie de funciones pequeñas que tienen entradas y salidas, y realizan una tarea específica [Zhu et al., 1997].

En el contexto de Prolog, las pruebas unitarias evalúan el desempeño de predicados específicos en el programa. El ejemplo del predicado `suma/3` presentado en la Sección

anterior sería una ilustración de pruebas unitarias. Para definir las pruebas unitarias de un módulo, primero es necesario analizar las diferentes estructuras en el código y documentar su funcionamiento. Así, se garantiza que las pruebas unitarias cubran todas sus funciones.

La tabla siguiente muestra los predicados pertenecientes al módulo `utils`, junto con el formato de sus argumentos tal como se describe en la documentación. Basándose en esta información, se desarrollarán los casos de prueba.

Función	**Argumentos**
rearrange/4	(?list, +term, +term, ?list)
univ_to/2	(?list, ?list)
univ/3	(?term, ?list, ?term)
call_over_file/4	(+string, +atom, +atom, ?term)
read_terms/2	(+stream, -list)
write_terms/2	(+stream, +list)
transpose/2	(+list, -list)

Tabla 6.1: Predicados y argumentos para el módulo utils

Estos predicados no tienen por qué ser los únicos en el fichero. Por ejemplo, el predicado `utils:transpose/2` utiliza otros tres predicados añadidos para procesar sus resultados. Sin embargo, desde la perspectiva de la lógica del programa, estos actúan en conjunto para producir un resultado medible y, por lo tanto, se consideran como una unidad a la hora de escribir las pruebas.

La función `utils:rearrange/4` se describe en detalle en el apartado 6.2.3. A partir de esta definición, se presentan a continuación los casos de prueba unitarios para el predicado, de manera simplificada, como ejemplo del resto.

```
% casos de uso habituales
test(rearrange) :-
    rearrange([1-a, 2-b, 3-c], A-B, B-A, [a-1, b-2, c-3]).

test(rearrange, [true(R == [a-1, b-2, c-3])]) :-
    rearrange([1-a, 2-b, 3-c], A-B, B-A, R).
test(rearrange, [true(R == [a-1, b-2, c-3])]) :-
    rearrange(R, A-B, B-A, [1-a, 2-b, 3-c]).

test(rearrange, [true(R == [1-_, 2-_, 3-_]), nondet]) :-
    rearrange(R, A-_, A, [1, 2, 3]).
test(rearrange, [true(R == [1-_, 2-_, 3-_]), nondet]) :-
    rearrange([1, 2, 3], A, A-_, R).

% comportamiento tolerado
test(rearrange, [nondet]) :-
    rearrange([1-a, 2-b, 3-c], _, 1, [1, 1, 1]).

test(rearrange, [true(R == [1-a, 2-a, 3-a])]) :-
    rearrange(R, A-a, A, [1, 2, 3]).
test(rearrange, [true(R == [1-a, 2-a, 3-a])]) :-
    rearrange([1, 2, 3], A, A-a, R).

test(rearrange, [true(N == 3), nondet]) :-
```

```
        rearrange([1, 2, 3], _, _, R),
        length(R, N).

    % situaciones limítrofes
    test(rearrange, [true(R == [])]) :-
        rearrange(R, A, A, []).
    test(rearrange, [true(R == [])]) :-
        rearrange([], A, A, R).
    test(rearrange, [true(R == [2,2,2])]) :-
        rearrange([1,1,1], 1, 2, R).
    test(rearrange, [nondet]) :-
        rearrange([1, 1, 1], 1, _, _).

    % casos de fallo
    test(rearrange, [fail]) :-
        rearrange([1-a, 2-b, 3-c], A-B, A-B, [a-1, b-2, c-3]).

    test(rearrange, [fail]) :-
        rearrange([1-a, 2-b, 3-c], _-B, B, [a, b, c]).
```

El objetivo de estas pruebas es validar las diferentes opciones que la documentación ofrece para el predicado. Las pruebas unitarias buscan lograr una cobertura completa de las variaciones del problema. En este contexto, se ilustra cómo se pueden utilizar las diferentes opciones de la herramienta desarrollada en este trabajo. Para entender este ejemplo, es esencial conocer si se han considerado todas las posibles instanciaciones de las variables, o lo que es lo mismo, la cobertura "ground" del predicado. Al ejecutar la herramienta, obtenemos:

```
predicate
  rearrange/4
     ground                    coverage
      rearrange(g,g,g,ng)       4
      rearrange(g,g,ng,ng)      4
      rearrange(g,ng,g,g)       4
      rearrange(g,ng,ng,ng)     17
      rearrange(ng,ng,ng,g)     13
      rearrange(g,ng,ng,g)      12
```

Esto indica que se han considerado adecuadamente todas las posibles instanciaciones de las variables para validar el comportamiento esperado del predicado.

Pruebas de integración

Las pruebas de integración tienen como objetivo asegurar el correcto funcionamiento de las interacciones entre los distintos módulos y servicios de una aplicación. Para diseñar las pruebas de integración de la herramienta, se ha utilizado como referencia el listado de dependencias entre módulos, que se detalla en el apartado 5.1.

La tabla siguiente especifica los puntos de entrada y salida de cada módulo. Estos puntos pueden ser clasificados en:

- **Exportados.** Este predicado será utilizado por otros módulos.

- **Importados.** Este predicado se utiliza dentro del módulo y proviene de otro módulo externo.
- **Externos.** Este predicado es invocado por la persona usuaria o realiza una acción externa, como mostrar información en pantalla o modificar un archivo.

	loader	**instrumenter**	**launcher**	**visualizer**
plcov	load_kb/2 get_cov_and_clear/0 clear_dynamic/0			
loader		instrument/3	init/2 coverage/1 clean/0 stop/0	visualize/1
instrumenter			insert_launcher_calls/5	
launcher	process_term/4 log_term/5			

Tabla 6.2: Predicados públicos de cada módulo (filas) y el módulo que los utiliza (columnas)

Los predicados externos de la herramienta son:

- `plcov`. Todos los predicados en `plcov` están accesibles para consulta en su uso.
- `instrumenter`. El predicado `instrument/1` crea un archivo donde escribe los resultados de la instrumentación.
- `visualizer`. El predicado `visualize/1` muestra en pantalla los resultados de la cobertura que recibe.

Las pruebas realizadas a estos predicados se han efectuado de manera similar al ejemplo proporcionado para las pruebas unitarias. Sin embargo, su enfoque se centra inicialmente en definir la funcionalidad de la función y escribir pruebas en torno a ella, en lugar de focalizar el esfuerzo en detectar casos de fallo y situaciones límite. Esta abstracción del comportamiento específico para centrar el esfuerzo en las acciones que realiza es posible gracias a la confianza que proporcionan las pruebas unitarias. Esto se debe a que las sentencias que estos predicados consultan durante su ejecución han sido validadas mediante pruebas unitarias.

Pruebas de aceptación

Las pruebas de aceptación son pruebas operacionales donde se examinan los aspectos de la herramienta como producto final. A diferencia de pruebas anteriores, estas no se centran en una Sección independiente del programa ni en la conexión entre distintas herramientas, sino que evalúan una ejecución completa en el contexto de los casos de uso.

Para la realización de las pruebas de aceptación de esta herramienta, se empleó un programa simple de 6 predicados que amplía el que se presenta en el Capítulo 3, basado en un sistema de recetas e ingredientes. Las pruebas se llevan a cabo siguiendo los pasos definidos en los casos de uso específicos del programa, asegurando no solo la corrección de los resultados, sino también que la herramienta opere conforme a lo previsto.

Las pruebas de aceptación, así como otros métodos para evaluar aspectos no funcionales del código, se benefician significativamente de la inclusión de personas que encajan

con el perfil de usuaria en el proceso. Estas pueden probar flujos de trabajo sin interferencias externas, indicando si las opciones son intuitivas o si requieren ajustes. En ciertos casos, incluso pueden identificar errores que no se detectaron durante las pruebas unitarias.

Para el desarrollo de esta herramienta, no fue posible contar con un grupo de personas usuarias para realizar estas pruebas, por lo que se recurrió a los casos de uso como sustitutos. Prolog no es un lenguaje ampliamente utilizado en el ámbito profesional, y su uso en el ámbito académico es limitado. Por ello, encontrar a personas que se ajusten a los requisitos detallados en los casos de uso es una tarea complicada. Aunque no sea un producto comercial, cualquier herramienta se beneficiaría de estas pruebas, y la necesidad de realizarlas de manera adaptada representa una limitación de la herramienta.

Pruebas de portabilidad

La portabilidad del programa es una característica esencial, tenida en cuenta en sus casos de uso. Para su ejecución, el programa requiere únicamente una instalación de SWI-Prolog, lo que permite su portabilidad sin modificaciones a sistemas Linux, Windows y MacOS.

Uno de los propósitos de esta herramienta era diseñar un sistema fácilmente portable a otras implementaciones de Prolog. Esta característica se consideró durante el diseño del programa. Por ello, ninguna de las funcionalidades implementadas excede lo que se podría lograr con las especificaciones ISO de Prolog. Sin embargo, desarrollar un sistema completamente portable a diferentes distribuciones y que cumpla de forma estricta con los estándares ISO es una tarea que supera los recursos de este proyecto.

Existen varias complicaciones que afectan la portabilidad de este sistema. Resolver estas complicaciones manteniéndose dentro de los estándares no tiene una solución directa:

- **Predicados no estándares ineficientes.** Aunque la implementación realizada busca limitar el uso de predicados de SWI-Prolog que no están incluidos en los estándares, aún hace uso de predicados pertenecientes a ciertas librerías comunes. Reimplementarlos directamente en el programa sería redundante, ineficiente y propenso a errores. Predicados como `append/3` y `length/2` son fundamentales para la gestión de listas en cualquier entorno de desarrollo, por lo que se emplean en esta implementación a pesar de no formar parte del estándar ISO.
- **Colisiones.** Dada la naturaleza del programa como una herramienta cercanamente relacionada con el funcionamiento interno de Prolog, es susceptible a tener colisiones con términos que no pertenecen al estándar ISO, pero que están definidos en otras implementaciones de Prolog. Un claro ejemplo es un error de colisión que ocurre cuando se utiliza esta herramienta en GNU-Prolog (otra implementación popular de Prolog), donde hay un conflicto con un predicado que ya existe con el mismo nombre.

Por estas razones, la portabilidad entre diferentes intérpretes en la herramienta se logra a través de características de diseño que facilitan la realización de cambios en el código. Esto permite crear, de forma ágil, una versión adaptada a cualquier sistema.

CAPÍTULO 7
Conclusiones

7.1 Objetivos alcanzados

Se han logrado con éxito los objetivos propuestos para este trabajo, cubriendo los casos de uso detallados y desarrollando una aplicación plenamente funcional y utilizable.

La herramienta presenta la información de forma clara y concisa a través de la interfaz de comandos. Esto permite ejecutar la cobertura de una consulta mediante un solo comando en la terminal o dos ejecuciones de predicados en la consola de SWI-Prolog.

La herramienta puede realizar mediciones de los distintos tipos de predicados definidos en el estándar ISO. Además, puede medir la cobertura de todos los predicados de un archivo sin necesidad de procesar la consulta introducida.

La herramienta soporta la ejecución de múltiples consultas y permite visualizar la cobertura en cualquier momento. Asimismo, puede medir la cobertura de predicados que alteran la búsqueda de soluciones con operaciones como "cut", "fail" y "call".

Ofrece tres modos de medición para analizar diferentes aspectos de la cobertura. En ellos, se puede observar tanto la cobertura de unificación de cláusulas como la instanciación de argumentos.

Además, la herramienta proporciona una experiencia de usuario transparente. Una vez activada en la consola de Prolog, las consultas habituales mostrarán automáticamente una medición de cobertura, generando un ambiente de pruebas eficiente e intuitivo.

El desarrollo de esta herramienta no solo se ha enfocado en el logro de estos objetivos, sino también en el aprendizaje profundo de las diversas tecnologías que integran Prolog y otros lenguajes de programación. Desde esta perspectiva, se considera que el proyecto ha sido exitoso, ya que ha explorado diferentes implementaciones del problema, aportando solidez a la solución final y un profundo entendimiento académico del lenguaje.

7.2 Relación con los estudios

El desarrollo de este proyecto está fuertemente relacionado con los conceptos abordados en las materias de programación de la carrera, tales como:

- Programación (PRG)
- Lenguajes, Tecnologías y Paradigmas de la programación (LTP)
- Estructuras de Datos y Algoritmos (EDA)

La asignatura LTP es especialmente relevante, pues proporciona una visión general de los lenguajes declarativos y del proceso de compilación, ambos aspectos fundamentales en la funcionalidad de esta herramienta.

En el ámbito declarativo, las estructuras de Prolog se asemejan a las demostraciones matemáticas, y el pensamiento declarativo se asemeja al empleado en el desarrollo de axiomas y definiciones matemáticas. A lo largo del proyecto, he descubierto que las definiciones y prácticas aprendidas en las materias de matemáticas de la carrera me ofrecieron herramientas valiosas para entender y diseñar programas Prolog con una estructura más eficiente. Estas materias son:

- Matemática Discreta (MAD)
- Teoría de Autómatas y Lenguajes Formales (TAL)

Para llevar a cabo este proyecto, apliqué numerosos conocimientos obtenidos a lo largo de la carrera, relacionados con las distintas etapas de un proyecto, como la definición de casos de uso, la aplicación de una metodología y la división de tareas, entre otros aspectos. El contenido relacionado con la organización y el diseño global de proyectos se aborda en materias como:

- Ingeniería del Software (ISW)
- Gestión de Proyectos (GPR)

Además de los conceptos abordados en estas asignaturas, ha sido necesario profundizar en las siguientes materias:

- El lenguaje Prolog. Para aprender este lenguaje, se han seguido los ejercicios y la teoría propuesta en los libros "Learn Prolog Now!" [Blackburn et al., 2006] y "PROLOG: Programming for Artificial Intelligence" [Bratko, 2001]. Adicionalmente, se han revisado y probado diversas bases de código en Prolog utilizadas a lo largo del desarrollo, así como algunas secciones del estándar ISO.
- La creación de un intérprete. Con el objetivo de garantizar un entendimiento más profundo del lenguaje y realizar una prueba funcional de una de las soluciones propuestas, se desarrolló un intérprete en el lenguaje Python para ejecutar programas en Prolog. Este intérprete es capaz de procesar programas simplificados y realizar consultas sobre ellos.
- La interfaz por consola y su interacción con SWI-Prolog. Para desarrollar un programa ejecutable a través de la línea de comandos, se investigó el material existente desde un punto de vista funcional con la herramienta de ejecución. También se estudió desde una perspectiva teórica para determinar qué requisitos debería cumplir una aplicación de línea de comando.

La realización de este trabajo ha representado un desafío. El lenguaje Prolog introduce notables diferencias con respecto al paradigma imperativo. Implementar una solución extensa en él para un programa que requiere un conocimiento avanzado de su estructura y proceso de interpretación y búsqueda ha sido complejo. Este desarrollo ha exigido establecer nuevas directrices de diseño basadas en otros paradigmas, una evaluación meticulosa de las funciones creadas y soluciones creativas para abordar problemas complicados.

7.3 Trabajo futuro

El desarrollo de la herramienta se ha llevado a cabo desde una perspectiva exploratoria, en consonancia con la metodología y el plan de trabajo adoptados. Esta aproximación ha consistido tanto en explorar el entorno de Prolog como en enfocarse en desarrollar una herramienta que cumpla con los objetivos propuestos. A pesar de que esta estrategia ha prolongado la duración del desarrollo, debido a la falta de conocimientos previos sobre el lenguaje y los entornos a utilizar, ha aportado resultados positivos al proyecto final.

Debido a este enfoque exploratorio, algunas ideas que surgieron durante el desarrollo y que excedían considerablemente los objetivos y el alcance del proyecto no se implementaron. A continuación, se presentan algunas de estas posibilidades para explorar futuras extensiones o modificaciones de este trabajo.

El funcionamiento integrado de la herramienta en la consola, que permite una ejecución transparente, no puede acceder a las soluciones no deterministas de una consulta usando el operador ";". Para visualizar estas mediciones, es necesario usar un predicado de Prolog como `findall/3` que realiza múltiples consultas. Para mejorar la transparencia, se podría incorporar esta funcionalidad de Prolog dentro de este modo de ejecución, analizando los puntos de decisión del programa y ajustando su funcionamiento según sea necesario.

Actualmente, la herramienta y otras similares mencionadas en el estado del arte, no miden la cobertura sobre los predicados internos de Prolog consultados durante la ejecución. Sin embargo, instrumentando el código, sería posible medir estos predicados de la misma manera que los del archivo, usando predicados intermedios que se consulten antes de dirigir la solución al predicado real. Esta opción podría ofrecer métricas valiosas al analizar la cobertura, especialmente en pruebas de rendimiento. Un desafío sería cubrir las consultas dinámicas durante la ejecución, ya que el uso de lanzaderas no contemplaría estas llamadas.

El diseño modular de la herramienta tiene como objetivo minimizar la cantidad de predicados necesarios para añadir funcionalidades adicionales y otras tareas de mantenimiento. En la misma línea, la tabla que muestra los resultados se ha diseñado de forma modular y extensible. Llevado al límite, este diseño modular podría permitir la introducción de predicados externos para el análisis de cobertura, ajustando así la medición a parámetros específicos que desee evaluar. Esto potenciaría enormemente la configurabilidad de la herramienta, permitiendo realizar extensiones sin modificar el código fuente.

Bibliografía

[Abrahamsson et al., 2017] Abrahamsson, P., Salo, O., Ronkainen, J. and Warsta, J., 2017. *Agile Software Development Methods: Review and Analysis*. arXiv preprint arXiv:1709.08439 [cs.SE].

[Apt, 1997] Apt, K.R., 1997. *From logic programming to Prolog* (Vol. 362). Prentice Hall, London.

[Blackburn et al., 2006] Blackburn, P., Bos, J., and Striegnitz, K., 2006. *Learn prolog now!* (Vol. 7, No. 7). College publications, Londres.

[Bratko, 2001] Bratko, I., 2001. *Prolog programming for artificial intelligence*. Pearson Education.

[Brna et al., 1991] Brna, P., Brayshaw, M., Bundy, A., Elsom-Cook, M., Fung, P., and Dodd, T., 1991. *An overview of Prolog debugging tools*. Instructional Science, 20, pp.193-214. Springer.

[Covington et al., 2009] Covington, M.A., Bagnara, R., O'Keefe, R.A., Wielemaker, J., and Price, S., 2009. *Coding Guidelines for Prolog*. CoRR, abs/0911.2899. Disponible en: `http://arxiv.org/abs/0911.2899`. Fecha de consulta: 15/08/2023.

[Free Software Foundation, 2007] Free Software Foundation, Inc., 2007. *GNU Lesser General Public License. Version 3*, 29 Junio 2007. Disponible en: `https://www.gnu.org/licenses/lgpl-3.0.en.html`. Fecha de consulta: 15/08/2023.

[Hammad and Inayat, 2018] Hammad, M. and Inayat, I., 2018. *Integrating risk management in scrum framework*. En: 2018 International Conference on Frontiers of Information Technology (FIT), pp.158-163. IEEE.

[IEEE, 1998] IEEE, 1998. *IEEE Recommended Practice for Software Requirements Specifications*. IEEE Std 830-1998, pp.1-40. doi: `10.1109/IEEESTD.1998.88286`.

[Jackson, 2012] Jackson, D., 2012. *Software Abstractions: logic, language, and analysis*. MIT Press.

[Martin, 2008] Martin, R.C., 2008. *Clean Code: A Handbook of Agile Software Craftsmanship (1st. ed.)*. Prentice Hall PTR, USA.

[Mills, 1976] Mills, H.D., 1976. *Software Development*. IEEE Transactions on Software Engineering, SE-2(4), pp.265-273. doi: `10.1109/TSE.1976.233831`.

[Moura, 2009] Moura, P., 2009, July. *From Plain Prolog to Logtalk Objects: Effective Code Encapsulation and Reuse*. En ICLP (p. 23).

[Peled, 2001] Peled, D.A., 2001. *Software Reliability Methods*. Springer.

[Pereira and Shieber, 2002] Pereira, F.C.N. and Shieber, S.M., 2002. *Prolog and natural-language analysis*. Microtome Publishing.

[Project Management Institute, 2021] Project Management Institute, 2021. *PMBOK GUIDE. The Standard for Project Management*. Disponible en: `https://www.pmi.org/pmbok-guide-standards/foundational/pmbok?sc_camp=D750AAC10C2F4378CE6D51F8D987F49D`. Fecha de consulta: 15/08/2023.

[Rentsch, 1982] Rentsch, T., 1982. *Object oriented programming*. ACM Sigplan Notices, 17(9), pp.51-57. ACM New York, NY, USA.

[Schimpf and Shen, 2012] Schimpf, J. and Shen, K., 2012. *ECLiPSe–from LP to CLP*. Theory and Practice of Logic Programming, 12(1-2), pp.127-156.

[Schwaber and Beedle, 2001] Schwaber, K. and Beedle, M., 2001. *Agile software development with Scrum*. Prentice Hall PTR.

[Sommerville, 2016] Sommerville, I., 2016. *Software Engineering, Global Edition*. Pearson Education. Disponible en: `https://books.google.es/books?id=W_LjCwAAQBAJ`. Fecha de consulta: 15/08/2023.

[Stallman, 2002] Stallman, R., 2002. *Free software, free society: Selected essays of Richard M. Stallman*. Lulu.com.

[Taylan et al., 2020] Taylan, O., Bafail, A.O., Abdulaal, R.M. and Kabli, M.R., 2020. *Project, program and portfolio management in large Dutch corporations—differences and similarities between PERT and CPM*. International Journal of Managing Projects in Business, 13(2), pp.214-236.

[Wielemaker et al., 2012] Wielemaker, J., Schrijvers, T., Triska, M., and Lager, T., 2012. *Swi-prolog*. Theory and Practice of Logic Programming, 12(1-2), pp.67-96. Cambridge University Press.

[Zhu et al., 1997] Zhu, H., Hall, P.A.V., and May, J.H.R., 1997. *Software unit test coverage and adequacy*. ACM Computing Surveys (CSUR), 29(4), pp.366-427. ACM New York, NY, USA.

APÉNDICE A
ODS

Grado de relación del trabajo con los Objetivos de Desarrollo Sostenible (ODS).

Objetivos de Desarrollo Sostenible	**Alto**	**Medio**	**Bajo**	**No procede**
ODS 1. **Fin de la pobreza.**				X
ODS 2. **Hambre cero.**				X
ODS 3. **Salud y bienestar.**				X
ODS 4. **Educación de calidad.**		X		
ODS 5. **Igualdad de género.**			X	
ODS 6. **Agua limpia y saneamiento.**				X
ODS 7. **Energía asequible y no contaminante.**				X
ODS 8. **Trabajo decente y crecimiento económico.**				X
ODS 9. **Industria, innovación e infraestructuras.**	X			
ODS 10. **Reducción de las desigualdades.**				X
ODS 11. **Ciudades y comunidades sostenibles.**				X
ODS 12. **Producción y consumo responsables.**				X
ODS 13. **Acción por el clima.**				X
ODS 14. **Vida submarina.**				X
ODS 15. **Vida de ecosistemas terrestres.**				X
ODS 16. **Paz, justicia e instituciones sólidas.**				X
ODS 17. **Alianzas para lograr objetivos.**				X

Reflexión sobre la relación del TFG/TFM con los ODS y con el/los ODS más relacionados.

Este trabajo se relaciona con varios Objetivos de Desarrollo Sostenible (ODS). En primer lugar, aborda el ODS 9 "Industria, innovación e infraestructuras". Este objetivo persigue el desarrollo de infraestructuras resilientes, promueve una industrialización inclusiva y sostenible, y fomenta la innovación. En el contexto del trabajo realizado, la herramienta de cobertura de Prolog contribuye de manera principal a este ODS.

La medida de cobertura de consultas aplicada a un programa en Prolog posibilita una detección de errores más refinada y un diseño mejorado de los casos de prueba. Esto lleva a un código de mayor calidad y a un producto software más optimizado. La calidad del software es un componente fundamental para las infraestructuras tecnológicas resilientes y robustas, ya que se requiere de validación y entornos estables para construir sobre herramientas ya existentes, y así desarrollar software de mayor complejidad y con usos más especializados. Además, en el desarrollo del trabajo, se analizan soluciones existentes al problema propuesto y se formula una solución innovadora que utiliza una técnica diferente para abordar la cobertura de código mediante la instrumentación. Esto también refuerza los conceptos del ODS 9, cuyo eje central es la innovación.

El ODS 4 "Educación de calidad" también es un componente importante de este trabajo. Este objetivo promueve la educación como motor de progreso e igualdad y como elemento esencial para el desarrollo sostenible.

El lenguaje de programación Prolog es una herramienta académica destacada y sigue siendo la principal en la enseñanza de programación lógica. El desarrollo en este proyecto de una herramienta que facilite la comprensión e implementación de programas en Prolog contribuye a mejorar este aspecto de la educación, facilitando el proceso de aprendizaje. El acceso a herramientas más intuitivas y mejor equipadas puede resultar en un aprendizaje más profundo y en una formación más robusta.

Se ha incluido también el ODS 5 "Igualdad de género", que busca eliminar todas las formas de discriminación contra las mujeres en el mundo.

A lo largo de este documento, se ha hecho un esfuerzo por utilizar un lenguaje inclusivo. La informática es uno de los campos digitales e ingenieriles donde la presencia masculina predomina. Por ende, se busca impulsar la participación de las mujeres en el ámbito informático mediante el uso del lenguaje inclusivo, informar a la sociedad sobre su presencia en este campo y destacar la importancia de sus contribuciones en el desarrollo de la informática.

Es importante destacar que se ha intentado equilibrar el uso del lenguaje inclusivo con la economía del lenguaje, evitando una lectura tediosa o repetitiva, optando por conceptos neutrales y colectivos que incluyan a ambos géneros.