UNIVERSITAT POLITÈCNICA DE CATALUNYA
BARCELONATECH
Facultat d'Informàtica de Barcelona

FIB

Cómo solucionar problemas de una PS2

Práctica Diagnóstico

Coneixement i Raonament Automàtic

Inteligencia Artificial

Autores

Pablo Barrenechea Perea

Artur Aubach Altes

Grupo 11

Profesor Ramon Sangüesa Sole

Cuatrimestre Primavera 2022/2023

CONTENIDOS

1. INTRODUCCIÓN

Este trabajo tiene como objetivo proponer una solución a un problema específico que pueda presentarse en la vida cotidiana, mediante un programa basado en un pequeño sistema experto de clasificación que realice la tarea cognitiva de diagnóstico. Durante su elaboración, se utilizaron herramientas como PROLOG y DIAGRAMS.NET para la creación de gráficos de árbol, facilitando la comprensión y aplicación del sistema propuesto. El trabajo incluye todo el código necesario para su implementación directa.

1.1. DIAGNÓSTICO

El diagnóstico es un proceso cognitivo que busca establecer las causas que pueden explicar la aparición de un conjunto de síntomas. A menudo, se agrupan clases de síntomas y causas y se establecen las relaciones más probables entre ellas. Este proceso se lleva a cabo mediante un ciclo de recolección y comprobación de información que describe el estado del problema a diagnosticar, y en el que se utiliza el conocimiento de dominio, expresado mediante reglas, para relacionar los síntomas observados con el estado de diagnóstico o conclusión. Este conocimiento puede ser obtenido tanto a partir de la experiencia de profesionales del ámbito del problema, como mediante técnicas de aprendizaje automático o una combinación de ambas.

1.2. PROLOG

Prolog es un lenguaje de programación lógico que se utiliza para desarrollar programas de inteligencia artificial y sistemas expertos. En lugar de utilizar un enfoque basado en reglas, como el lenguaje de programación convencional, Prolog utiliza un enfoque basado en la lógica matemática y el razonamiento para resolver problemas. Es un lenguaje declarativo, lo que significa que el programador describe qué debe hacer el programa en lugar de cómo debe hacerlo. Esto permite a Prolog trabajar de manera muy eficiente con problemas complejos que requieren un gran procesamiento de información.

En este trabajo, utilizaremos el lenguaje de programación lógico Prolog para desarrollar todo el código necesario para el sistema experto de clasificación. Esto incluirá la creación de

reglas y la implementación de algoritmos para realizar la tarea de diagnóstico cognitivo. Prolog es una herramienta adecuada para este tipo de trabajo debido a su enfoque basado en la lógica y su capacidad para manejar problemas complejos de inferencia.

1.3. DIAGRAMS.NET

DIAGRAMS.NET es una herramienta en línea para la creación y edición de diagramas y gráficos. Su interfaz intuitiva y las numerosas opciones de diseño disponibles hacen que sea una opción ideal para la creación de diagramas de flujo, diagramas de procesos, diagramas de red, diagramas de organización y muchos más. Además, ofrece integración con Google Drive y Microsoft OneDrive, lo que permite trabajar en equipo en tiempo real y colaborar con colegas y compañeros de manera sencilla. DIAGRAMS.NET es una herramienta esencial para cualquiera que necesite crear diagramas y gráficos de alta calidad de manera rápida y eficiente.

En este proyecto, utilizaremos DIAGRAMS.NET como una herramienta complementaria para facilitar la comprensión y visualización del dominio abordado por nuestro programa. El uso de gráficos y diagramas en DIAGRAMS.NET ayudará a clarificar los aspectos del problema y del proceso de diagnóstico, lo que mejorará la comprensión del proyecto.

2. DOMINIO

Creemos que para poder hacer un buen programa de diagnosis, es muy importante definir previamente qué dominio queremos tratar y cuál queremos que sea su alcance, puesto que esto nos ayudará tanto a entender con claridad lo que queremos que haga nuestro programa, como a agilizar el proceso de creación.

2. 1. ELECCIÓN DEL DOMINIO

Aprovechando que podemos elegir el dominio del programa, dentro de un abanico muy amplio de posibilidades, reflexionar para saber cuál puede ser la mejor opción, no es una pérdida de tiempo, puesto que la elección de un mal dominio, puede conducir hacia un trabajo muy enrevesado y *ambiguo*.

2. 1. 1. CRITERIOS DE ELECCIÓN

A nuestro parecer, un buen dominio es aquel, que sea fácil tanto para el usuario a la hora de definir los síntomas del problema, como para el programa interpretar esos síntomas. En otras palabras, reducir al máximo la *inexactitud* causada por la *incertidumbre* y la *imprecisión* o *vaguedad*.

Asimismo, tratar un tema del cual no estamos familiarizados, tampoco ayuda.

En consecuencia, elegiremos ese dominio en que todos los participantes del grupo *dominen y se sientan cómodos*, las preguntas que haga el programa para identificar el problema estén *exentas de interpretación* y la respuesta del usuario sean preferiblemente *booleanos*.

2. 1. 2. DOMINIO ELEGIDO

Repasando la lista de dominios, podemos identificar tres grandes ramas:

- **Diagnóstico de salud:** Poca familiarización + Posibilidad de interpretación + Impreciso + incierto.
- **Diagnóstico de equipamientos de música:** Familiarización nula + ?.
- **Diagnóstico de averías:** Familiarización en algunos campos + Poco subjetivo + Frecuencia de booleanos.

Aplicando nuestros criterios en la lista de dominios, podemos observar que la rama más adecuada debería ser el "Diagnóstico de averías".

Finalmente, haciendo un análisis más extenso observando de qué trata cada dominio, hemos elegido: <Cómo solucionar problemas de una PS2>

2. 2. DESCRIPCIÓN DEL DOMINIO

Una vez hemos elegido el dominio que queremos tratar, es hora de definir su alcance.

El dominio aborda soluciones de posibles problemas que puedan aparecer al usar una PS2. Para detectar qué problema tienes y poder ofrecerte la mejor solución, hace preguntas objetivas, que espera como respuesta un Sí/No.

Como hemos mencionado en el apartado "CRITERIOS DE ELECCIÓN", todo esto son características del que consideramos para nosotros un buen dominio.

Como podemos observar en la web, el dominio consta de 4 grandes apartado, o lo que es lo mismo, soluciona 4 posibles problemas que te pueda suceder en tu PS2:

- **Si la consola no enciende**
- **Si los controladores no funcionan**
- **Si un juego no funciona**
- **Si la bandeja del juego no se abre**

2. 1. 1. SI LA CONSOLA NO ENCIENDE

En esta rama, el programa te dará una solución si el problema que tú tienes es, que tu consola PS2 no enciende.

Como podemos observar en la Figura 1, el dominio es capaz de dar solución hasta 3 motivos por el cual su PS2 no enciende, además esta rama no tiene posibilidad de salirse del dominio, en otras palabras, siempre dará una solución si su problema es alguno de estos 3 motivos.

Somos conscientes que existen más motivos por los cuales una PS2 no se enciende, sin embargo, recuerdo que nos estamos ciñendo a la información que nos proporciona la web.

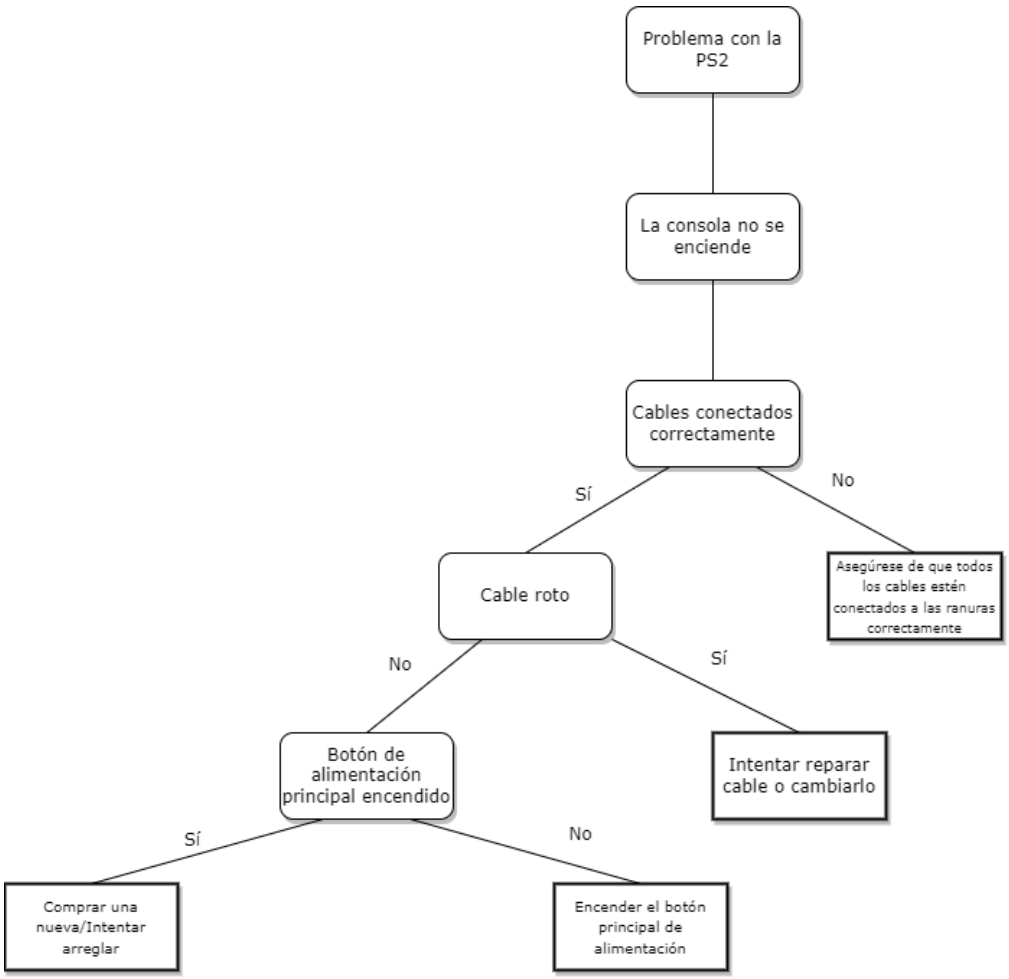

Figura 1. Árbol, La consola no se enciende.

2. 1. 2. SI LOS CONTROLADORES NO FUNCIONAN

En esta rama trataremos posibles motivos por los cuales su controlador PS2 no funciona. Las dimensiones de la rama, son exactamente las mismas que en "Si la consola no enciende". Puesto que da solución hasta 3 motivos del problema, sin salirse del dominio. Figura 2.

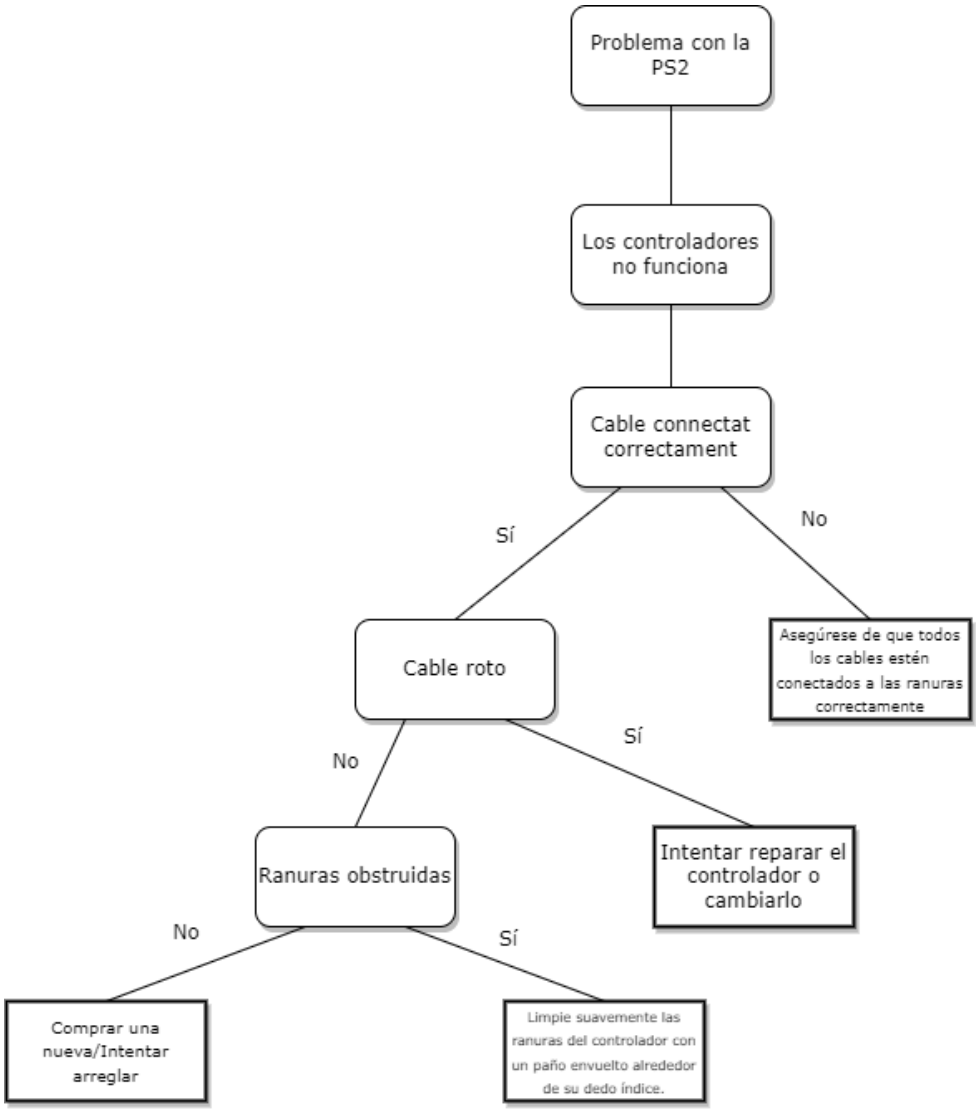

Figura 2. Árbol, Los controladores no funcionan.

2. 1. 3. SI UN JUEGO NO FUNCIONA

Si tu algún juego de la PS2 deja de funcionar, esta branca posiblemente te dé la solución, esta rama cabe recalcar que sí puede quedarse fuera del dominio:

Problema con la PS2 → El disco está sucio → Poniendo un disco de limpieza, sigue yendo mal (NO)

Sin embargo, no estamos muy convencidos de las posibles soluciones que da la Web, ya que no soluciona los problemas más comunes que te puedan suceder relacionados con el disco, como por ejemplo:

- Si mi disco está rayado?
- Solo no me funcionan los discos azules/lilas, eso quiere decir algo o puedo hacer algo al respecto?

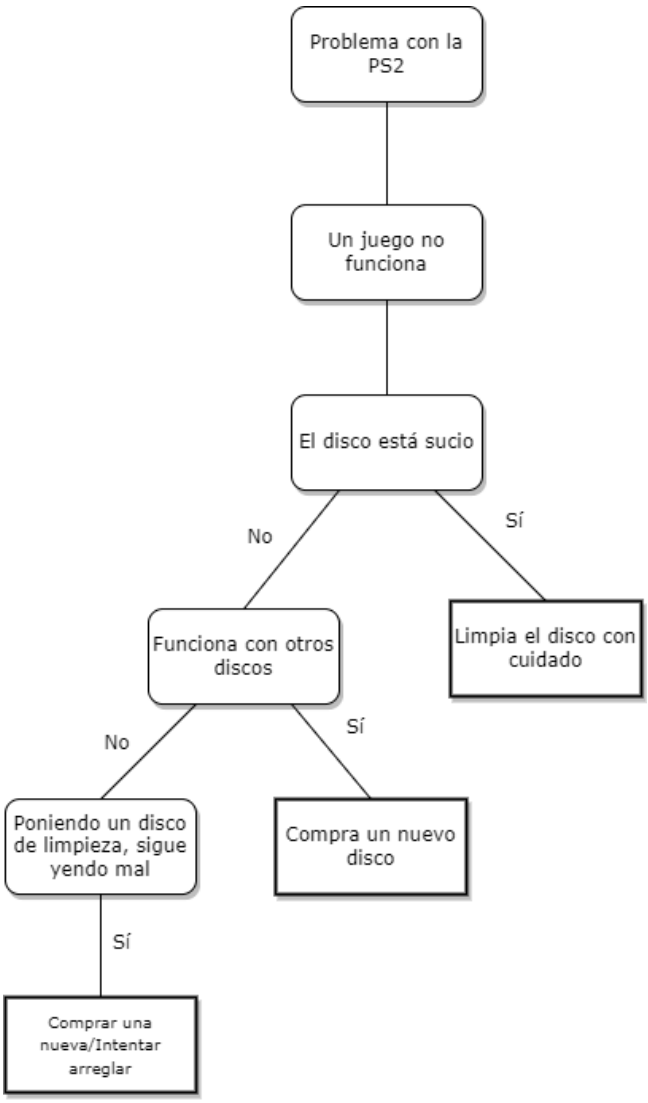

Figura 3. Árbol, Un juego no funciona.

2. 1. 4. SI LA BANDEJA DEL JUEGO NO SE ABRE

Por último, trataremos el problema de sí su bandeja de juegos de la PS2 no se abre. Como podemos observar en la Figura 4, aquí tenemos la novedad de que existen preguntas que a la misma vez pueden ser la solución del problema.

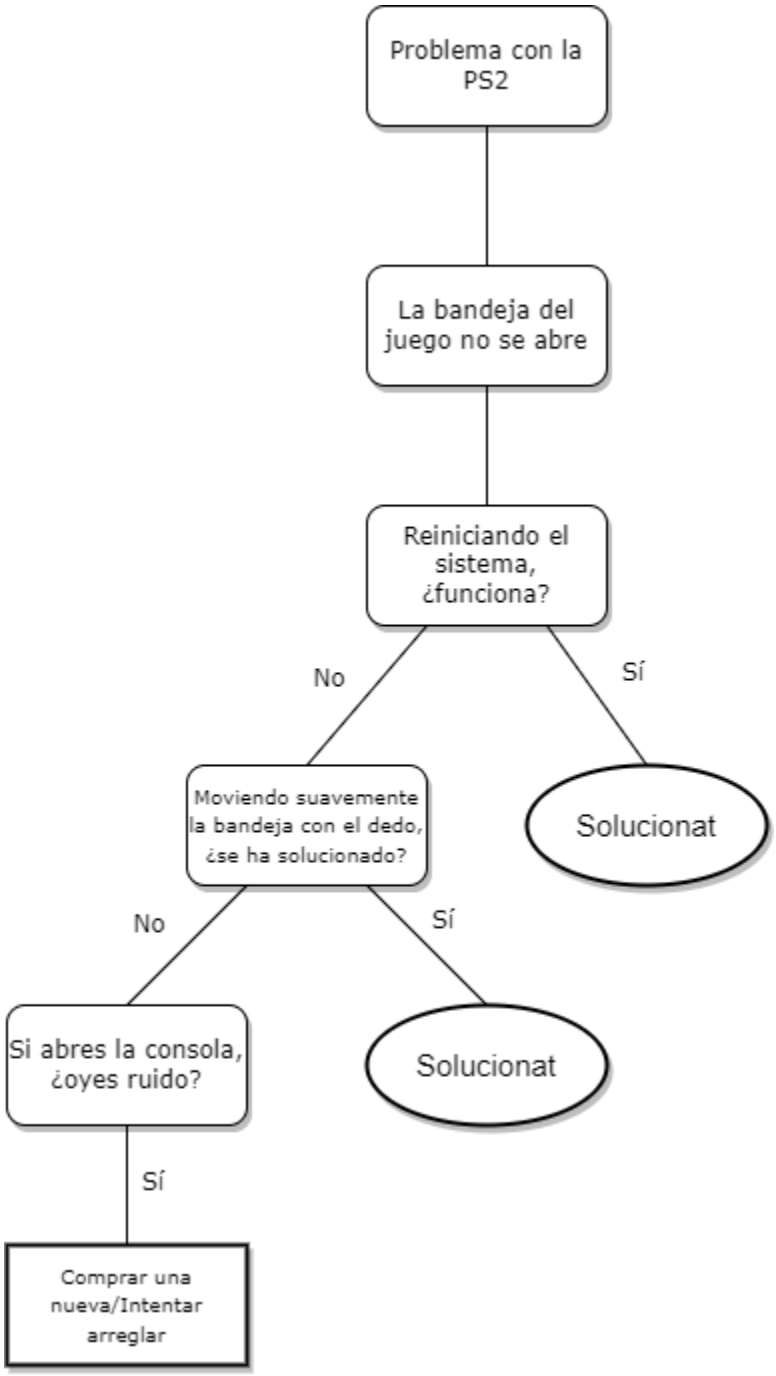

Figura 4. Árbol, La bandeja del juego no se abre.

2. 1. 5. CONJUNTO DEL DOMINIO

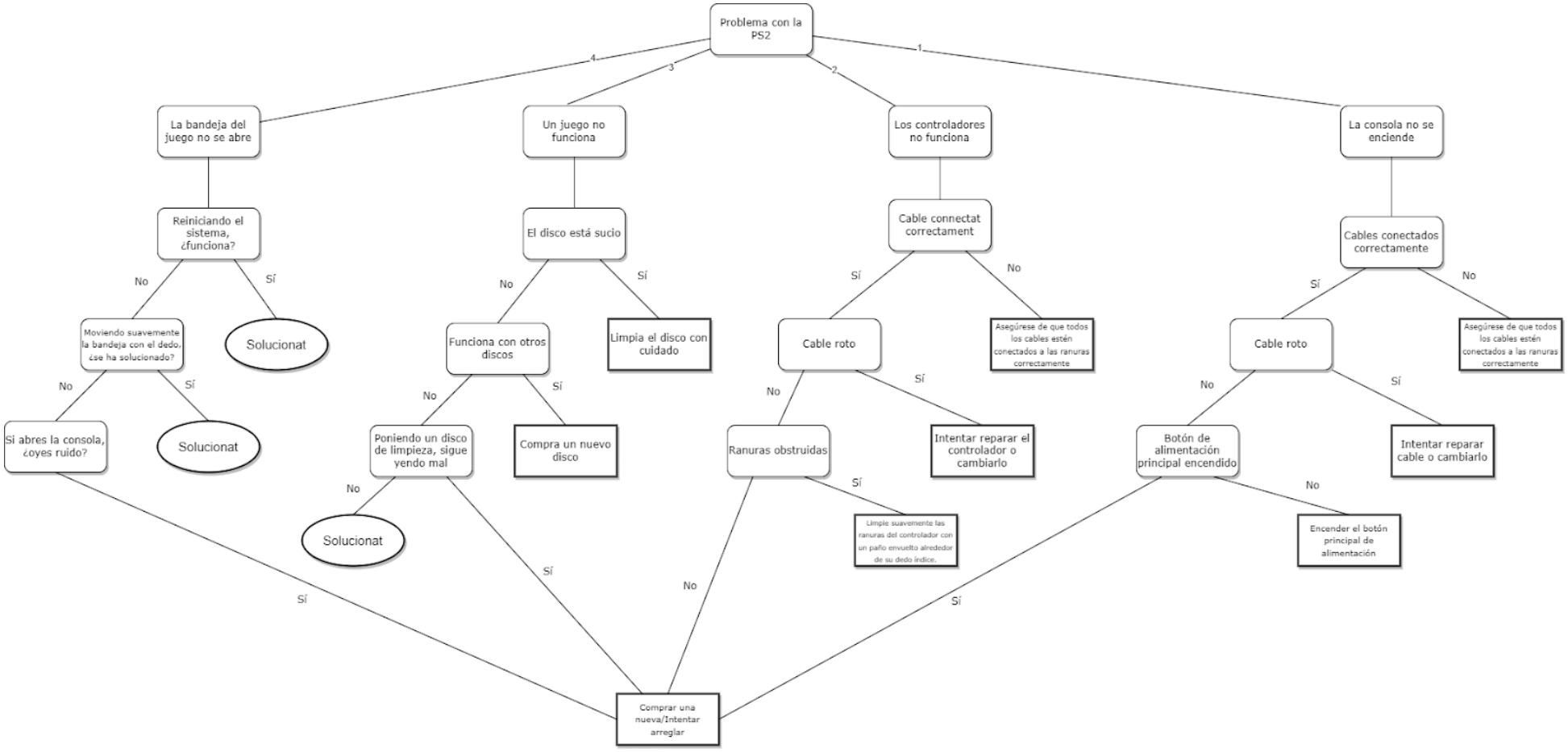

Figura 5. Árbol, Conjunto del dominio.

2. 3. AMPLIACIÓN/MODIFICACIÓN DEL DOMINIO

Como hemos mencionado antes, no nos convencía las soluciones que daba la Web del dominio para sí "un juego no funciona". Por eso nos hemos visto obligados a ampliar y modificar esta rama, con información de otras Webs y dar una solución más amplia y sólida.

Buscando por distintos sitios web, hemos encontrado dos páginas las cuales nos daba solución a lo que creemos que es para nosotros, los motivos más frecuentes por los cuales, un juego deja de funcionar: <How to Fix a PS2 Disc Read Error>, <Make PS2 Purple Discs Read in Your PS2 Simply by Using Tape>. Figura 6.

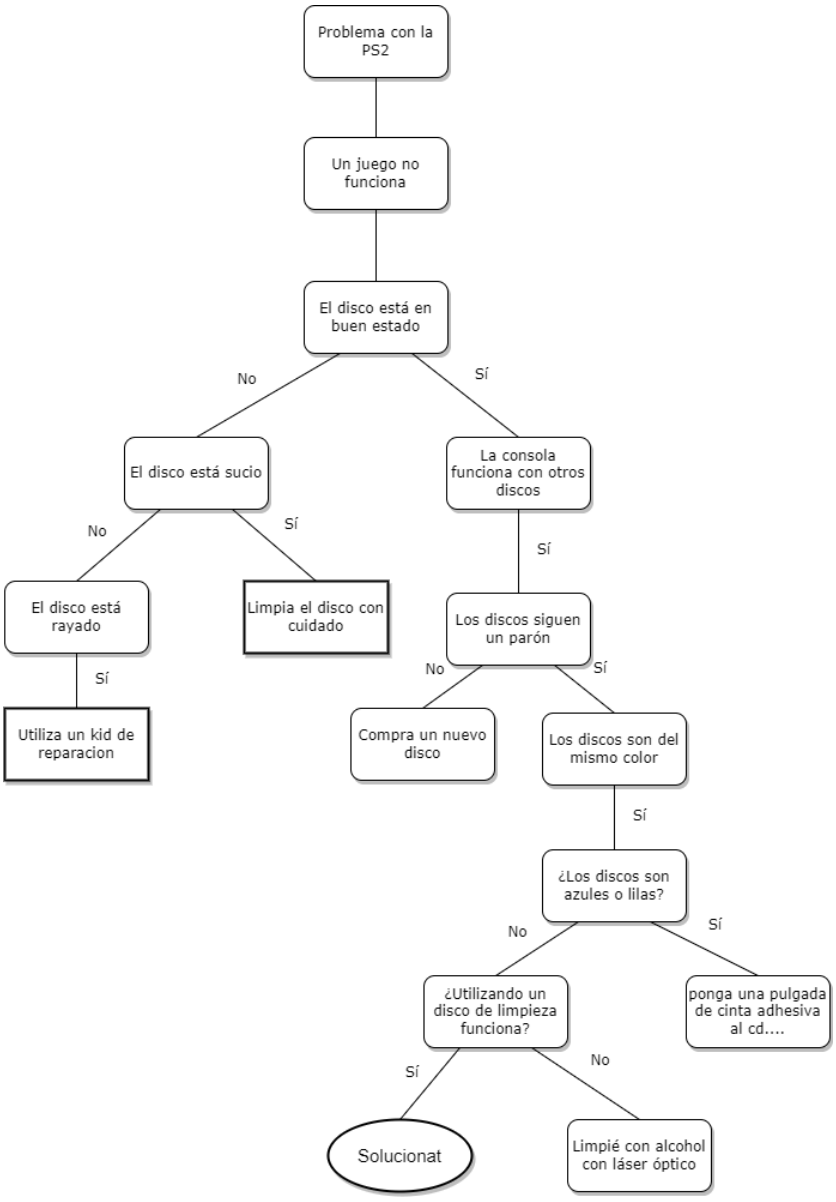

Figura 6. Árbol, Ampliación/Modificación del dominio.

2. 4. DOMINIO FINAL

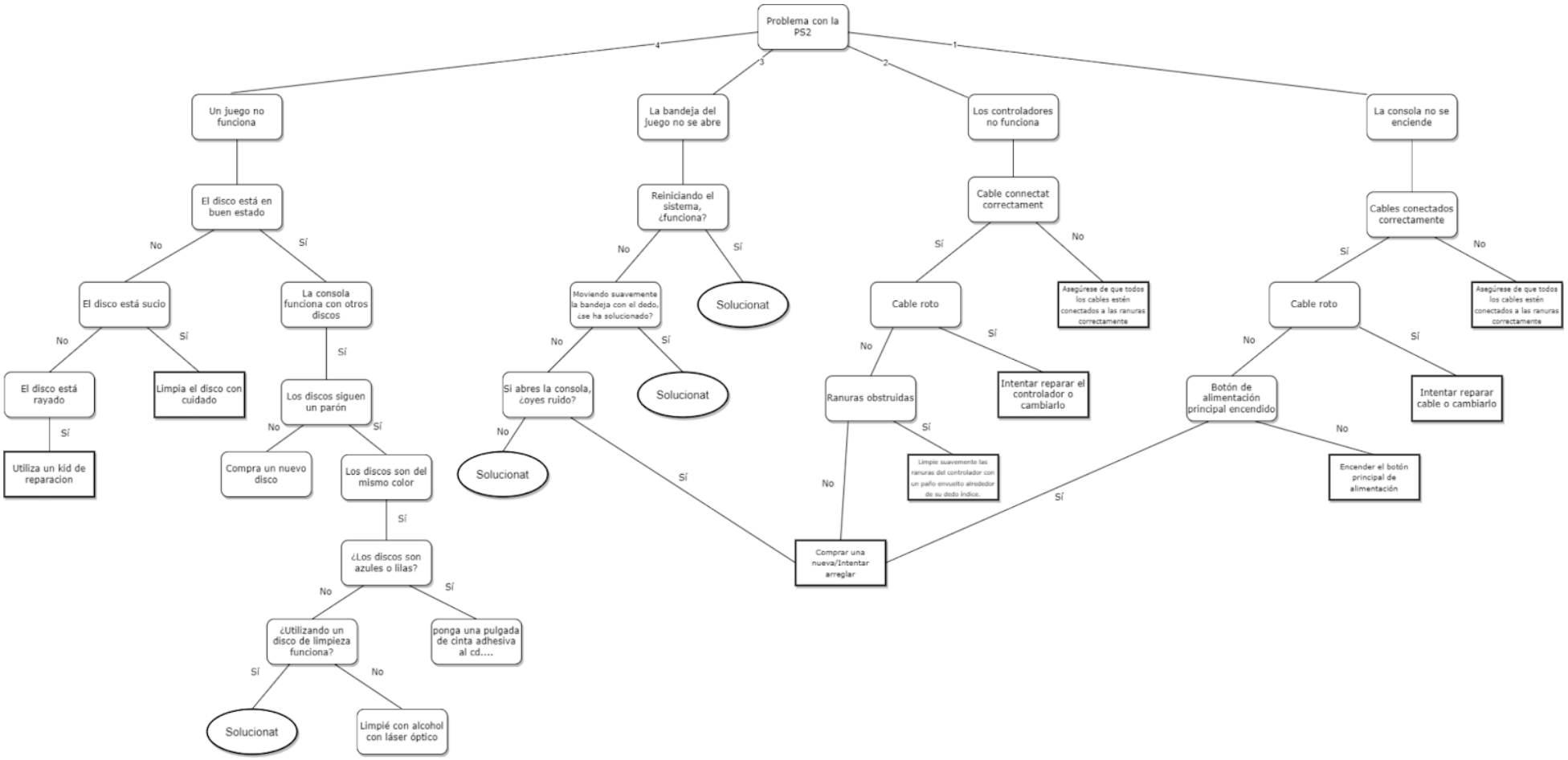

Figura 7. Árbol, Dominio final.

3. FORMALIZACIÓN DEL CONOCIMIENTO DEL DOMINIO

Elegido y estructurado ya el dominio, el siguiente paso en el proyecto viene a ser traducir dicho dominio a distintas normas y hechos que introduciremos después en nuestro programa. Así pues, basando nuestra arquitectura del programa en el ya estudiado programa: "identificacio_animals.pl", dividimos el dominio en las siguientes categorías:

- Hipótesis
- Hechos

Además, a estas piezas del puzzle que sería el dominio, teníamos que agregarles una serie de funciones que modificaran los estados de dichas piezas, pudiendo así deducir un puzzle distinto para cada situación que el usuario nos presente. Dichas funciones quedan resumidas en:

- Función de negación afirmación
- Función de verificación
- Función de pregunta
- Función de explicación

Dadas las bases de lo que viene a ser nuestro programa, pasemos a ver más a fondo cómo hemos traducido las reglas del dominio, y el propio dominio, de un lenguaje que nosotros entendemos a uno que entienda el programa prolog.

3. 1. LOS HECHOS Y LAS HIPÓTESIS

Llamaremos hecho a todo aquello que preguntemos al usuario, con el objetivo de que este nos vaya confirmando o negando dichos hechos. Estos hechos serán también los que irán concretando el problema que tiene aquello que estemos estudiando (en nuestro caso la consola), llegando al final a hacer un diagnóstico.

Este diagnóstico es el que conoceremos como hipótesis. Llamaremos al diagnóstico así debido a la naturaleza del prolog. Pues dado que prolog funciona encadenando una serie de funciones y hechos hasta que consigue que todos ellos le cuadren, la mejor manera de conseguir que este resuelva un problema es haciendo que este haga una suposición y vaya comprobando que todos los hechos que han de cumplirse para que se cumpla dicha hipótesis son ciertos. Así pues, a la que uno de los hechos de esa hipótesis no se cumpla, prolog descartará la hipótesis y pasará a la siguiente.

Definidos ambos términos, podemos pasar a traducir nuestro conocimiento del dominio del lenguaje y los diagramas a un programa que prolog pueda ejecutar.
Comenzaremos por las hipótesis. Para cada hipótesis crearemos una función:

hipotesis(“introducir_hipótesis”)

Tal que, en el caso de que la consola no encienda (Figura 1), como ejemplo, la hipótesis intentar_reparar_cable_o_cambiarlo, se cumplirá siempre que se cumplan los hechos indicados en el dominio. En este caso, los hechos que han de cumplirse serán:

- La consola no se enciende
- Cable roto

Así pues, nos quedaría lo siguiente:

La_consola_no_se_enciende ^ Cable_roto → intentar_reparar_cable_o_cambiarlo

Dado este ejemplo, podemos pasar a concluir la fórmula general para una hipótesis, que será la siguiente:

hechoA1 ^ hechoA2 ^ … ^hechoAn → hipotesis(A)

Conociendo que en prolog la implicación va de derecha a izquierda y se utiliza el símbolo: **:-** para ello, nos quedará la siguiente definición para la función hipótesis:

hipotesis(a) :- hecho_a_1; hecho_a_2; … ; hecho_a_n

o, en el caso del ejemplo:

hipotesis(*intentar_reparar_cable_o_cambiarlo*) **:-** *la_consola_no_se_enciende* **;**
*cable_roto***.**

Dicho esto, la única diferencia entre lo anterior y nuestro programa, es que para la comodidad a la hora de leerlo, en vez de definir la hipótesis de ese modo, primero le damos un apodo, y después a dicho apodo le agregamos los hechos que han de cumplirse para llegar a la hipótesis:

hipotesis(a) :- apodoA1
(. . .)
apodoA1 :- hecho_a_1; hecho_a_2; … ; hecho_a_n

o volviendo al ejemplo:

hipotesis(*intentar_reparar_cable_o_cambiarlo*) **:- apodo_E_1**

apodo_E_1 :- *la_consola_se_enciende* **;**
*cable_roto***.**

3. 2. El USUARIO Y LA AFIRMACIÓN/NEGACIÓN DE UN HECHO

Una vez entendido como hacer que un programa sepa que solución habrá que dar conocidos una serie de hechos, el siguiente paso es determinar qué hechos se han dado y cuáles no. Es aquí donde entrará el papel del usuario. El usuario será el puente que una el conocimiento que le hayamos introducido al programa junto el mundo real, certificando y determinando una serie de hechos que conseguirán que el programa llegue a una conclusión.

Para comunicarnos con el usuario, utilizaremos las funciones del prolog:

- read(*Variable que se unificará con aquello que el usuario introduzca*)
- write(*String que se imprimirá en la pantalla*)

Además, crearemos una serie de funciones que nos ayuden a distinguir los hechos que sí han sucedido frente a los que no:

- pregunta(*Mensaje*)
- verifica(*Mensaje*)
- si(*Mensaje*)
- no(*Mensaje*)

3. 2. 1. PREGUNTA

La función pregunta es la principal encargada de recoger la información del usuario. Lo que queremos que está función haga es preguntar al usuario si se da uno de los hechos que el programa tiene en su base de datos, con el objetivo de ir acotando las posibilidades hasta llegar a una conclusión. Así pues, principalmente, queremos que esta función escriba una pregunta del estilo "**hecho** + *?*" por pantalla, y después recoja la respuesta del usuario. En prolog, esto se puede programar de la siguiente manera:

```
pregunta( Mensaje ) :- write ( Mensaje ),    %Escribe el hecho que tenga por variable
                       write (' ? '),        %y un símbolo de pregunta.
                       read( Respuesta ).    %Finalmente, guarda en la variable
                                             Respuesta lo que el usuario introduzca.
```

Siguiendo con el ejemplo anterior, el usuario vería esta pantalla:

```
?- se_enciende_la_consola ?
|:                          /*Aquí iría la respuesta del usuario*/
```

Ahora, aquello que el usuario nos conteste, queremos transformarlo en conocimiento para el programa, pues por mucho que el usuario responda si, el programa aún no sabe si ese hecho se está cumpliendo o no.

3. 2. 2. AFIRMACIÓN Y NEGACIÓN DE UN HECHO

Teniendo en mente el objetivo de transformar una respuesta que el usuario nos de en conocimiento para el programa, lo primero que se nos viene a la cabeza es pensar en el tipo de pregunta que haremos, para saber el tipo de respuestas que deberíamos esperar. Como las preguntas van a ser de sí o no, las respuestas también lo serán, ahorrándonos así el tener que entender lo que el usuario escriba; ya que, tan sólo necesitamos que nos conteste un sí o un no.

Sin embargo, por mucho que el usuario responda sí o no, hay que buscar una manera de traducirle dicha respuesta al programa. El método más sencillo es crear una función si/1, tal que el argumento de la función **si/1** sea un hecho que sí sucede. Por otro lado, la función **no/1**, determinará aquellos hechos que no sucedan.

Dicho esto, lo único que quedaría es comprobar si la respuesta ha sido sí o no; y según la respuesta, crearemos una nueva función **si**(*Hecho*) o **no**(*Hecho*) para cada hecho que preguntemos. Cabe destacar el detalle de que en el caso que la respuesta sea sí, se establecerá **si**(*Hecho*) y no se realizarán más acciones, devolviendo **pregunta**(*Hecho*) **true**. Sin embargo, en el caso de que la respuesta sea no, además de establecer no(Hecho), deberemos incluir el comando fail, con el objetivo de que el valor de pregunta(Hecho) sea **false** (esto se debe a que así, al llegar a la hipótesis, el hecho sea falso y la hipótesis no podrá confirmarse, eliminándola de las candidatas. Esto quedará más claro más adelante).

Por último, hay que saber que para que una función pueda ser creada mientras el programa está en ejecución, habrá que declararla dinámica (con la función **dynamic**) y habrá que utilizar una función que la agregue para cada variable distinta (la función **assert/1**). El código quedará así:

```
pregunta( Mensaje ) :- write ( Mensaje ),
                       write (‘ ? ‘),
                       read( Respuesta ),

                       (Respuesta == si
                       -> assert( si( Mensaje ) );
                              (assert( no( Mensaje ) ) , fail )).
```

/* La flecha (->) se encarga de hacer si(*Mensaje*) en caso de que Respuesta == si, y en el contrario, creará no(*Mensaje*). Finalmente, faltaría declarar si y no como funciones dinámicas: /*

```
:- dynamic si/1, no/1.
```

Con esto ya tendríamos un pequeño programa que preguntaría si un hecho sucede, y registraría la respuesta; pero aún sigue faltando una pieza clave en este puzzle: una función que verifique si el hecho sucede o no, y que en caso de que aún no se haya registrado dicho hecho, pregunte al usuario por ello.

3. 2. 3. VERIFICA

La función **verifica/1** se encargará de realizar lo propuesto en el anterior apartado: verificar si un hecho sucede o no, y preguntar al usuario por ello en caso de que aún no se haya registrado la afirmación/confirmación de dicho hecho. Para ello, hemos de tener en cuenta el funcionamiento del prolog, que se basa en evaluar distintas cláusulas hasta llegar a una en la que sea cierto lo evaluado, o evaluará todas las cláusulas sin encontrar ningún true y devolverá un false.

Sabiendo esto, la función **verifica** que vamos a construir devolverá true en caso de que el hecho suceda (es decir, existe **si**(*hecho*)), y false en caso contrario. Así pues, la función que estamos intentando construir quedará del siguiente modo:

```
verifica( Hecho ) :- si( Hecho )
                        -> true ;
                           ( no( Hecho ),
                             -> fail;
                                pregunta( Hecho )).
```

El código comprobará si **si**(*Hecho*) existe, y en ese caso, dará true, confirmando que el hecho se cumple. En caso contrario, evaluará **no**(*Hecho*). Si existe, hará fail, haciendo que la verificación sea falsa, y si no existe **no**(*Hecho*), realizará la función **pregunta**(*Hecho*).

3. 2. 4. HECHOS, HIPÓTESIS, PREGUNTA Y VERIFICA

Construidas ya las funciones básicas para el programa, podemos entender el funcionamiento al juntarlo todo en uno. Comencemos por las hipótesis:

La hipótesis, como ya hemos mencionado previamente, consta de comprobar que todos los hechos que el dominio indique que han de cumplirse para que la hipótesis se cumplan se cumplen. O expresado en prolog:

```
hipotesis( hipotesis ) :- verifica( hecho1 ), … , verifica( hechoN ).
```

En el caso de que el programa evalúe una hipótesis, llamará a la función **verifica** del primer hecho que haya en la lista de hechos. En caso de que no esté ya definido, llamará a la función pregunta, y depende de la respuesta, la función pregunta será **false** (hará fail hasta llegar al **verifica**(*hechoX*) o **true** (hará **Exit** hasta llegar a **verifica**(*hechoX*)). Así pues, al ser sí la respuesta, el programa continuará con la hipótesis, evaluando **verifica**(*hechoX+1*) o, en caso de que hechoX fuera el último, devolviendo la hipótesis que se esté evaluando. En el caso contrario, la hipótesis será fallida, pues al estar compuesta por conjunciones de hechos, en el momento que uno sea falso, toda la hipótesis será falsa.

Lo único que quedaría es crear más de una hipótesis, cada una con distintos hechos, de modo que el programa vaya evaluando una a una, y descartándolas según el usuario le diga que se cumple o no los hechos que esté evaluando.

3. 2. 5. NEGACIÓN Y AFIRMACIÓN II

Sin embargo, nuestro conocimiento del dominio incluye algo más, pues hemos tenido en cuenta que habrá preguntas a las que al responder no, el usuario estará afirmando que un hecho sí se da. Como ejemplo, está el caso del hecho "se_enciende_la_consola". Si el usuario responde no, está en una manera, confirmando el hecho "NO_se_enciende_la_consola", que es el hecho necesario para entrar su rama, tal y como lo expresa la Figura 7.

Es por ello que hemos modificado las funciones **verifica/1** y **pregunta/1**, pasando estas a ser **verifica/2** y **pregunta/2**. El objetivo de este segundo argumento es el de determinar si la respuesta que confirme un hecho que sea clave para determinar una hipótesis es la palabra no o sí. Así pues, para todos los hechos semejantes al anterior caso, modificaremos la función **verifica/2** y **pregunta/2** tal que, en el caso de **verifica/2**, si dicho argumento es no, al encontrar **no**(*Hecho*) devolverá **true**, y al encontrar **si**(*Hecho*) devolverá un **fail**. Por otro lado, la función pregunta funcionará tal que si su segundo argumento es no y el usuario le responde no, hará **assert**(**no**(*Hecho*)). En cambio, si el usuario le responde sí, hará **assert**(**si**(*Hecho*)) y **fail**. Realizadas estas modificaciones, ambas funciones quedarán de la siguiente manera:

```
verifica( Hecho, Confirmar_hecho) :-
        ( (Confirmar_hecho == si),
        si( Hecho) -> true ;
                ( no( Hecho) -> fail;
                        pregunta( Hecho, Confirmar_hecho) )  ) ;

        ( (Confirmar_hecho == no),
        no( Hecho) -> true ;
                ( si( Hecho) -> fail;
                        pregunta( Hecho, Confirmar_hecho) )  ) .


pregunta( Mensaje, Confirmar_hecho ) :- write ( Mensaje ),
                        write (' ? '),
                        read( Respuesta ),

                        ( Confirmar_hecho == si ,
                        (Respuesta == si -> assert( si( Mensaje ) );
```

(**assert**(**no**(*Mensaje*)) **, fail**)) **);**

(*Confirmar_hecho* == *no*,
(*Respuesta* == *no*-> **assert**(**no**(*Mensaje*) **);**
(**assert**(**si**(*Mensaje*)) **, fail**)) **) .**

Dadas ya estas definiciones de funciones, ya podemos construir nuestro programa que sepa realizar diagnósticos en función de respuestas de sí o no por parte de un usuario.

3. 3. FORMALIZACIÓN DEL DOMINIO ENTERO

Explicadas ya las hipótesis y funciones básicas, podemos comenzar a formalizar nuestro dominio completo, pues a pesar de que parezca sencillo a primera vista, hay que tener en cuenta una serie de excepciones y asegurarse de que todo se formaliza de manera correcta.

3. 3. 1. SI EL ORDENADOR NO SE ENCIENDE

La primera "rama" de nuestro dominio que formalizaremos será aquella cuyas hipótesis tengan un hecho en común: que el ordenador no se encienda. Basándonos en el diagrama del apartado dominio (Figura 1), lo primero que llama la atención es que, a pesar de que para entender bien el dominio, las ramas se vayan siguiendo una a una según lo que conteste el usuario, estos no son los hechos que confirmarán las hipótesis. Por ejemplo, para que la hipótesis "encienda_el_botón_de_alimentación" se cumpla, no hace falta que el cable esté roto ni desconectado; con que la consola no se encienda y el botón de encendido esté apagado, es suficiente.
Es por ello, que para visualizar los hechos que hacen falta para que se confirme una hipótesis, utilizaremos este otro diagrama (Figura 8). Como se puede observar, si escogemos cualquiera de las hipótesis y vamos hacia la izquierda siguiendo las flechas, cada bloque por el que se pase será uno de los hechos necesarios para que se cumpla dicha hipótesis.

Dicho esto, ya podemos comenzar a formalizar todo el dominio. Lo único que debemos tener en cuenta es que hechos han de cumplirse para que una hipótesis se confirme, si la respuesta para que cada hecho se confirme ha de ser sí o no, y, finalmente, en qué orden queremos que el programa los pregunte (según la Figura 7, pues no tendría mucho sentido que pregunten si funcionan los controladores en caso de que la consola no se encienda).

Para la primera rama, nos quedará lo siguiente:

hipotesis(*asegurese_de_que_los_cables_estan_correctamente_conectados_a_las_-ranuras***)** **:-** *hipotesis_E_1*.

hipotesis(*intente_reparar_el_cable_o_compre_uno_nuevo***)** **:-** *hipotesis_E_2*.

hipotesis(*enienda_el_boton_de_alimentacion_principal***)** **:-** *hipotesis_E_3*.

hipotesis_E_1 **:- verifica(***se_enceinde_la_consola***, no),**
verifica(*estan_correctamente_conectados_todos_los_cables,* ***no*****).**

hipotesis_E_2 **:- verifica(***se_enceinde_la_consola***, no),**
verifica(*algun_cable_esta_roto***, si).**

hipotesis_E_3 **:- verifica(***se_enceinde_la_consola***, no),**
verifica(*esta_encendido_el_boton_de_alimentacion_principal,* **no).**

Como se puede apreciar, el prolog comprobará primero la hipótesis E1, preguntando si se enciende la consola, y en caso de que el usuario responda “no”, preguntando si los cables están correctamente conectados, etc. Siguiendo así el orden determinado en el dominio y el sentido común en cuanto a gramática.

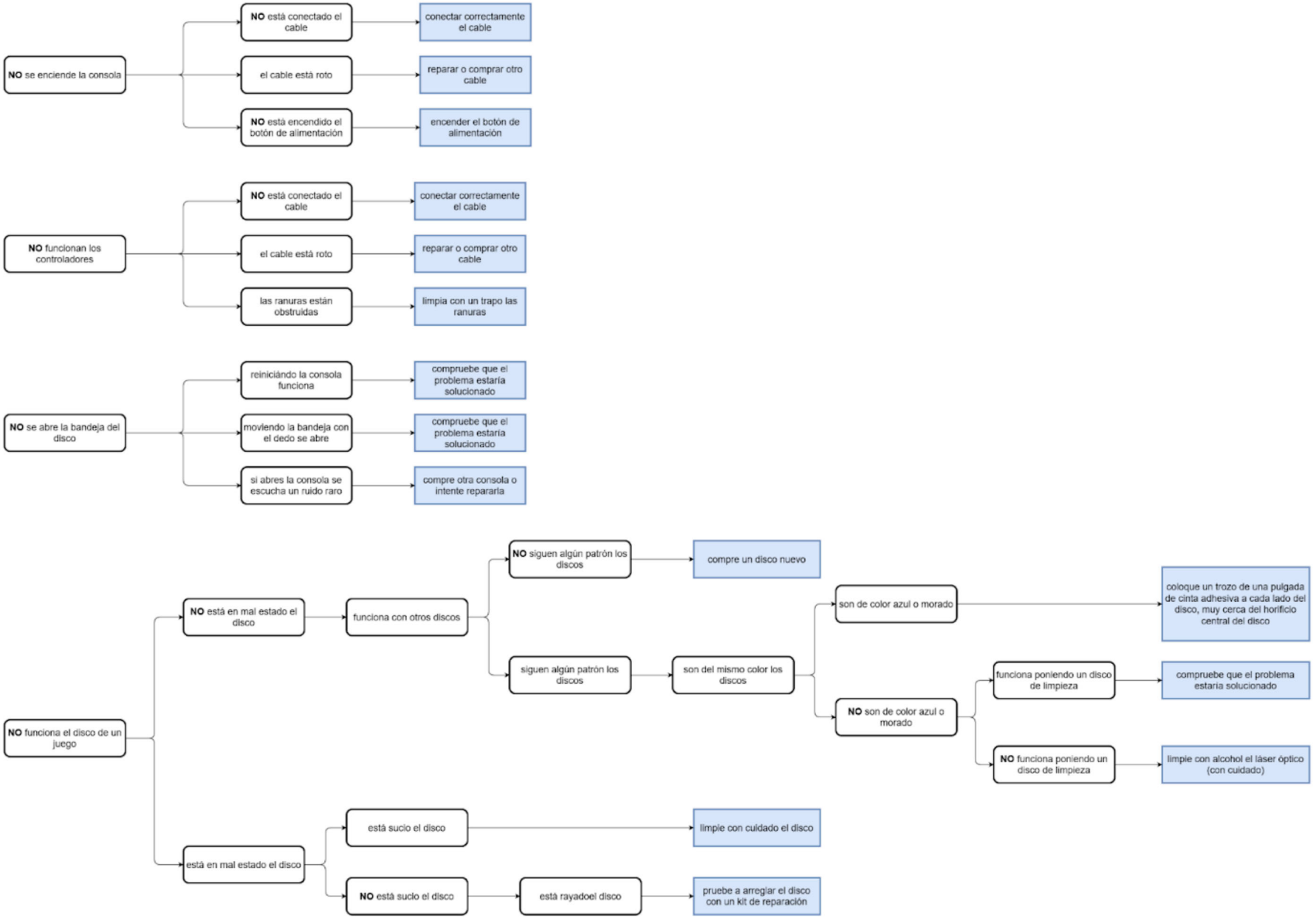

Figura 8.Diagrama de hechos y confirmación de hipótesis

3. 3. 2. NO FUNCIONAN LOS CONTROLADORES Y NO SE ABRE LA BANDEJA DEL DISCO

Para las ramas que estudian los distintos casos si el problema está relacionado con los controladores o con la bandeja del disco (Figuras 2 y 4 respectivamente), el comportamiento será el mismo que en la anterior, quedando las siguientes cláusulas:

hipotesis(*asegurese_de_que_los_cables_del_controlador_estan_correctamente_conectados-_a_las_ranuras***)** **:-** *hipotesis_C_1.*
hipotesis(*intente_reparar_el_cable_del_controlador_o_compre_uno_nuevo***)** **:-** *hipotesis_C_2.*
hipotesis(*limpie_con_cautela_las_ranuras_utilizando_un_trapo_envuelto_en_la_punta_del-_dedo_indice***)** **:-** *hipotesis_C_3.*

hipotesis(*compruebe_que_el_problema_estaria_solucionado***)** **:-** *hipotesis_S_1.*
hipotesis(*compruebe_que_el_problema_estaria_solucionado* **)**:- *hipotesis_S_2.*
hipotesis(*compre_otra_consola_o_intente_repararla_internamente***)** **:-** *hipotesis_S_3.*

hipotesis_C_1 **:- verifica(***funcionan_los_controladores***, no),**
verifica(*estan_correctamente_conectados_todos_los_cables_de_los_controladores,* ***no*****).**
hipotesis_C_2 **:- verifica(***funcionan_los_controladores***, no),**
verifica(*el_cable_del_controlador_esta_roto***, si).**
hipotesis_C_3 **:- verifica(***funcionan_los_controladores***, no),**
verifica(*las_ranuras_estan_obstruidas,* **si).**

hipotesis_S_1 **:- verifica(***se_abre_la_bandeja_del_juego***, no),**
verifica(*reiniciando_la_consola_se_abre_la_bandeja,* **si).**
hipotesis_S_2 **:- verifica(***se_abre_la_bandeja_del_juego***, no),**
verifica(*moviendo_la_bandeja_con_el_dedo_se_abre***, si).**
hipotesis_S_3 **:- verifica(***se_abre_la_bandeja_del_juego***, no),**
verifica(*se_escucha_un_ruido_raro_si_le_quitas_la_tapa_a_la_consola,* **si).**

3. 3. 3. NO FUNCIONA EL DISCO DE UN JUEGO

Finalmente, la formalización de la cuarta y última rama de nuestro dominio (Figura 6), difiere un poco de las anteriores tres. La mayor diferencia es que, como se puede apreciar en el diagrama que muestra los hechos necesarios para que una hipótesis se cumpla, hay hipótesis que necesitan de la confirmación positiva de un hecho y hay otros que necesitarán la confirmación negativa del mismo hecho. Como es en el caso de la hipótesis "limpie con cuidado el disco" y "compre un disco nuevo", que además de otros hechos, necesitan "no está en mal estado el disco" y "está en mal estado el disco" (respectivamente).

Así que, con el objetivo de ahorrarnos complicaciones, lo que haremos será agregar verifica(Hecho,no) en caso de que necesitemos una confirmación negativa, y verifica(Hecho,si) para los casos en los que se requiera una confirmación positiva de un hecho:

hipotesis(*compre_un_disco_nuevo***) :-** *hipotesis_J_1_1***.**

hipotesis(*coloque_un_trozo_de_una_pulgada_de_cinta_adhesiva_a_cada_lado_del_disco_-muy_cerca_del_horificio_central***) :-** *hipotesis_J_1_2_1***.**

hipotesis(*compruebe_que_el_problema_estaria_solucionado***) :-** *hipotesis_J_1_2_2***.**

hipotesis(*limpie_con_delicadeza_el_laser_optico_utilizando_alcohol***) :-** *hipotesis_J_1_2_3***.**

hipotesis(*limpie_con_cuidado_el_disco***) :-** *hipotesis_J_2_1***.**

hipotesis(*pruebe_a_arreglar_el_disco_con_un_kit_de_reparacion***)** *:- hipotesis_J_2_2***.**

hipotesis_J_1_1 **:- verifica(***funciona_el_disco_de_un_juego***, no),**
verifica(*esta_en_mal_estado_el_disco,* ***no*****),**
verifica(*funciona_con_otros_discos,* **si),**
verifica(*siguen_algun_patron_los_discos***,no).**

hipotesis_J_1_2_1 **:- verifica(***funciona_el_disco_de_un_juego***, no),**
verifica(*esta_en_mal_estado_el_disco,* ***no*****),**
verifica(*funciona_con_otros_discos,* **si),**
verifica(*siguen_algun_patron_los_discos***, si),**
verifica(*son_del_mismo_color_los_discos***, si),**
verifica(*son_de_color_azul_o_morado_los_discos***, si).**

hipotesis_J_1_2_2 **:- verifica(***funciona_el_disco_de_un_juego***, no),**
verifica(*esta_en_mal_estado_el_disco,* ***no*****),**
verifica(*funciona_con_otros_discos,* **si),**
verifica(*siguen_algun_patron_los_discos***,si),**
verifica(*son_del_mismo_color_los_discos***, si),**
verifica(*son_de_color_azul_o_morado_los_discos***, no),**
verifica(*funciona_despues_de_poner_un_disco_de_limpieza***, si).**

hipotesis_J_1_2_3 **:- verifica(***funciona_el_disco_de_un_juego***, no),**
verifica(*esta_en_mal_estado_el_disco,* ***no*****),**
verifica(*funciona_con_otros_discos,* **si),**
verifica(*siguen_algun_patron_los_discos***,si),**
verifica(*son_del_mismo_color_los_discos***, si),**
verifica(*son_de_color_azul_o_morado_los_discos***, no),**
verifica(*funciona_poniendo_despues_de_poner_un_disco_de_limpieza***, no).**

hipotesis_J_2_1 **:- verifica(***funciona_el_disco_de_un_juego***, no),**
verifica(*esta_en_mal_estado_el_disco,* ***si*****),**
verifica(*esta_sucio_el_disco***, si).**

hipotesis_J_2_2 **:- verifica(***funciona_el_disco_de_un_juego***, no),**
verifica(*esta_en_mal_estado_el_disco,* ***si*****),**
verifica(*esta_rayado_el_disco***, si).**

Y con esto habríamos terminado la clasificación de todas las hipótesis, excepto que aún falta un último detalle: ¿qué pasa si lo que el usuario dicta está fuera del dominio?

3. 4. LÍMITES DEL DOMINIO

Dadas ya todas las hipótesis y los hechos, podríamos pensar que ya está acabado el programa, sin embargo, ¿qué pasa con todas aquellas diferentes combinaciones de confirmaciones de hechos que no llevan a ninguna hipótesis?

Esta clase de hipótesis las denominaremos los límites de nuestro dominio, pero hemos de tener en cuenta, que no basta con expresar que está fuera del dominio y listo, sino que, basándonos en el dominio expresado en la Figura 8, habrá dos tipos de límites:

- Fuera del dominio
- Compre una consola nueva o intente repararla internamente

3. 4. 1. COMPRE UNA CONSOLA NUEVA O INTENTE REPARARLA

Como se puede apreciar en la Figura 7, en el caso de que el usuario entre dentro de alguna de las tres primeras ramas (pongamos que es la de la consola no se enciende), y no da confirmación de ningún hecho que lleve al programa a la confirmación de alguna hipótesis, en vez de tener que pasar a la siguiente rama, el programa tendría que dar la solución: compre una consola nueva o intente repararla internamente.

Para intentar formalizar esta "hipótesis" especial, nos aprovecharemos del funcionamiento del prolog. Prolog intenta unificar las variables con todas las posibilidades que encuentre, y si hay más de una, escogerá la que más arriba se encuentre en el programa. En este sencillísimo programa de ejemplo, prolog tiene una función joya a la que se le asigna el átomo amatista primero, y zafiro después:

joya(*amatista*).
joya(*zafiro*).

Así pues, si cargáramos el programa y hiciéramos:

?- **joya(X).**
X = *amatista* **;**
X = *zafiro***.**

Como se puede observar, lo primero con lo que prolog unifica la variable X es con amatista, pues se había declarado antes que zafiro. Este es el principio que utilizaremos para todas estas excepciones.

Lo que haremos será crear una hipótesis (**hipotesis**(*compre_una_consola_nueva_o_intente-_repararla_internamente*) y la situaremos justo al final de cada rama que tendría que desembocar en esta hipótesis, haciendo así, que se cumpla siempre y cuando tengamos la confirmación del hecho principal de la rama y se hayan negado las otras tres hipótesis. En el ejemplo de "la consola no se enciende", este será el hecho que querríamos comprobar; pues si nos negasen dicho hecho, quiere decir que el problema se encuentra en alguna otra rama, y no debería confirmarse la hipótesis "compre una consola nueva o intente repararla internamente".

Dicho esto, siguiendo el ejemplo de la rama “la consola no se enciende”, el código quedaría algo así:

hipotesis(*asegurese_de_que_los_cables_estan_correctamente_conectados_a_las_-ranuras***)** **:-** *hipotesis_E_1*.

hipotesis(*intente_reparar_el_cable_o_compre_uno_nuevo***)** **:-** *hipotesis_E_2*.

hipotesis(*enienda_el_boton_de_alimentacion_principal***)** **:-** *hipotesis_E_3*.

hipotesis(*compre_una_consola_nueva_o_intente_repararla_internamente***) :-**
verifica(*se_enciende_la_consola***, no).**

hipotesis_E_1 **:- verifica(***se_enceinde_la_consola***, no),**
verifica(*estan_correctamente_conectados_todos_los_cables,* ***no*****).**

hipotesis_E_2 **:- verifica(***se_enceinde_la_consola***, no),**
verifica(*algun_cable_esta_roto***, si).**

hipotesis_E_3 **:- verifica(***se_enceinde_la_consola***, no),**
verifica(*esta_encendido_el_boton_de_alimentacion_principal,* **no).**

De este modo, si el usuario responde “sí” a “se enciende la consola”, saltará a la siguiente rama. En cambio, si el usuario responde “no”, comprobará primero la hipótesis “E_1”, a continuación la “E_2”, después la “E_3” y, finalmente, escogerá la hipótesis de reparar la consola.

3. 4. 2. FUERA DEL DOMINIO

Tras haber visto el anterior caso, existe otra excepción a la que nuestro programa se ha de enfrentar, y es que, cabe la posibilidad de que el usuario haga una serie de elecciones tal que no exista hipótesis para responderle, lo que le llevaría al programa a fallar. Es por ello que, la última hipótesis que crearemos será aquella que el programa ofrecerá una vez haya probado todas las demás hipótesis.

Así pues, agregaremos dicha hipótesis al final del programa, haciendo que sea la última en comprobarse, y no necesite ninguna confirmación de ningún hecho para “confirmarse”. El código quedaría así:

< “todo el resto de las hipótesis” >

hipotesis(*fuera_del_dominio***) .**

Sin embargo, hay otro caso en el que se necesita informar que está fuera del dominio, cuando en la rama de la bandeja del disco, el usuario diga que se oye un ruido raro si abre la consola. Así que, está la trataremos como si de una hipótesis normal se tratase, introduciéndola con el resto de hipótesis de la rama:

< "las otras hipotesis(X) de la rama" >
hipotesis(*fuera_del_dominio***) :-** *hipotesis_S_4*
(…)
hipotesis_S_4 **:- verifica(***se_abre_la_bandeja_del_juego***, no),**
verifica(*se_escucha_un_ruido_raro_si_le_quitas_la_tapa_a_la_consola,* **no).**

Y con esto, ahora sí, todo nuestro conocimiento del dominio estaría formalizado a un programa de prolog que es capaz de diagnosticar el problema que tiene una playstation 2, según una serie de síntomas que el programa le preguntará al usuario, quien irá afirmando o negando dichos síntomas. Lo único que faltaría sería una función inicio/0, tal que esta llame a **hipotesis**(*Variable*) de una variable y después imprima la variable, reconociéndola como el resultado o la conclusión.

4. DAR UNA EXPLICACIÓN

Una explicación es básicamente el acto de guiarnos desde una serie de premisas hasta una conclusión, de manera que podamos razonar con más facilidad el cómo se ha llegado desde los puntos A1, A2, … , An, hasta la conclusión B. De hecho, la palabra proviene del latín "explicacion", que en su momento tenía el significado de desenvolver. Es por ello que ahora una explicación, es el acto de desenvolver la cadena de razonamientos que nos han llevado de A a B.

En el caso de nuestro programa, el siguiente paso a formalizar los conocimientos del dominio, es encargarse de que el programa sea capaz de explicarte cómo ha llegado desde tus respuestas a la conclusión. En el caso de este proyecto, es suficiente con que el programa muestre una lista de hechos que han sido necesarios para confirmar la hipótesis que ha seleccionado como respuesta. Así pues, estamos hablando de una función que vaya añadiendo elementos a una lista, siendo estos elementos las conclusiones que hayan sido confirmadas por el usuario.

4. 1. RAMAS 1, 2 Y 3

Para las hipótesis de las primeras tres ramas a las que accederá el programa (se enciende la consola, funcionan los controladores y se abre la bandeja de los discos) sabemos que no hay hechos con la asignación invertida. Esto es, si miramos el diagrama de la Figura 8, no encontraremos en ningún par de hipótesis A y B, dos hechos tal que:

A = hecho_a1 ^ … ^ hecho_X ^ … ^ hecho_aN

B = hecho_b1 ^ … ^ **NO** hecho_X ^ … ^ hecho_bN

Esto nos facilita mucho las cosas, pues con tal de que cada vez que se le haga una pregunta al usuario y este nos confirme algún hecho (ya sea con la respuesta sí o no), habrá que definir una función dinámica que nos guarde en la memoria que ese hecho tendrá que ser añadido a la lista. Eso no es tan complicado de pasar a código como pueda parecer, pues lo único que se debería hacer es modificar la función **pregunta/1** del siguiente modo:

```
pregunta( Mensaje, Confirmar_hecho ) :-
                write ( Mensaje ),
                write (‘ ? ‘),
                read( Respuesta ),

                ( Confirmar_hecho == si ,
                (Respuesta == si -> (assert( si( Mensaje )),
                                    assert( explicacion_si( Mensaje )) );
                                        (assert( no( Mensaje ) ) , fail )) );

                ( Confirmar_hecho == no,
                (Respuesta == no -> (assert( no( Mensaje )),
                                    assert( explicacion_no( Mensaje )) );
                                        (assert( si( Mensaje ) ) , fail )) ) .

:- dynamic si/1, no/1, explicacion_si/1, explicacion_no/1.
```

Como se puede apreciar, hemos añadido las funciones **explicacion_si/1** y **explicacion_no/1**, ya que a la hora de crear la lista de explicaciones, los enunciados o hechos a los que se haya contestado que no pero hayan confirmado un hecho (se enciende la consola, por ejemplo) habrá que añadirles una partícula negativa con el objetivo de que la explicación tenga sentido. Es decir, si el usuario contestase no a “se_enciende_la_consola ?” Y esto acabará derivando en la confirmación de una hipótesis, al crear la lista de explicación, no podríamos contestar: [… , se_enciende_la_consola, …]. Sino que, tendríamos que añadirle un no por delante: [… , no se_enciende_la_consola, …].

4. 2. RAMA 4: “NO FUNCIONA EL DISCO DE UN JUEGO”

Realizadas las modificaciones para las anteriores ramas, esta última tendrá algo más de complicación, pues al contrario de las tres primeras, en esta sí se dará la ocasión en la que dos hipótesis distintas con el mismo hecho, pero diferente asignación. Es decir se dará el caso de que suceda algo semejante a lo siguiente:

A = hecho_a1 ^ … ^ hecho_X ^ … ^ hecho_aN
B = hecho_b1 ^ … ^ **NO** hecho_X ^ … ^ hecho_bN

El caso es que esto supone un problema para nuestra implementación anterior, pues, tomando como ejemplo las hipótesis “limpie con cuidado el disco” y “repare el disco con un kit de reparación”, ambas necesitan el hecho “esta_sucio_el_disco”, sólo que la primera lo necesitará con un sí, y la segunda con un no. El caso es que la pregunta de dicho hecho sólo se realizará una vez, en la que se asignará si(Hecho) o no(Hecho). Y, como “limpie con cuidado el disco” será escogida antes, en caso de contestar con un no a esta pregunta, al haber asignado al segundo argumento de la función verifica un sí (porque recuerdo que estamos en la hipótesis de “limpie con cuidado el disco”), hará fail y pasará a la siguiente hipótesis (“repare el disco con un kit de reparación”). Hasta aquí iría todo bien, a continuación entraría en dicha hipótesis, y al realizar verifica con el segundo argumento no, encontraría no(esta_sucio_el_disco) y haría true, entrando al siguiente hecho.

Sin embargo, hay un detalle que nos hemos dejado. No se ha realizado **assert**(**explicacion_no**(*esta_sucio_el_disco*)), debido a que en el momento de hacer la pregunta, el segundo argumento era un sí y se ha respondido con un no, obviando dicho assert. Pero, como necesitamos la explicación de este hecho, habrá que buscar una solución.

La que nosotros hemos encontrado más sencilla era agregar un tercer argumento a las funciones **verifica/2** y **pregunta/2**, pasando ahora a ser **verifica/3** y **pregunta/3**. El objetivo de este tercer argumento es que al ser “si”, se accederá a un assert de **explicacion_si/no**(*Hecho*), para todos los casos de la tercera rama. Así pues, el único cambio en la función verifica es el añadir un argumento más para pasarlo después a pregunta al llamarla. El cambio grande es el que sucede en pregunta, en el que se añadirá el assert de la manera ya explicada:

```
pregunta( Mensaje, Confirmar_hecho, Doble_explicacion ) :-
        write ( Mensaje ),
        write (‘ ? ‘),
        read( Respuesta ),

        ( Confirmar_hecho == si ,
        (Respuesta == si -> (assert( si( Mensaje )),
                    assert( explicacion_si( Mensaje )) );
                        (assert( no( Mensaje ) ),
                        (Doble_explicacion == si
```

```
                                    -> (assert(explicacion_no( Mensaje )), fail) ;
                                        fail ))) );

            ( Confirmar_hecho == no,
            (Respuesta == no -> (assert( no( Mensaje )),
                            assert( explicacion_no( Mensaje )) );
                                (assert( si( Mensaje ) ),
                                (Doble_explicacion == si
                                -> (assert(explicacion_si( Mensaje )), fail) ;
                                    fail ))) );

:- dynamic si/1, no/1, explicacion_si/1, explicacion_no/1.
```

Hecho esto, ya tendremos todas las explicaciones necesarias para construir la lista con ellas.

4. 3. LISTA DE EXPLICACIONES

Ahora lo último que queda por hacer, es preguntarle al usuario si quiere una explicación y darle una lista con el conjunto de explicaciones, una vez le hayamos dado la confirmación de alguna hipótesis.

Para ello, lo primero que necesitaremos es una función que inicie el programa y de una respuesta, tal que:

```
inicio :- hypotesis( Hipotesis ),
        write('Quieres saber por qué?')
        read( Respuesta ),
        Respuesta == si -> explicacion( Hipotesis );
                        nl.
```

Después necesitaremos una función explicación/1, tal que esta se encargue de escribir la lista de manera aceptable:

```
explicacion( Hipotesis ) :- write(  Hipotesis ),
                            write("porque, "),
                            validar_porque( Lista ),
                            write( Lista ).
```

validar_porque(*[Cabeza | Cola]* **) :- (si(** *Cabeza* **),**
explicacion_si(*Cabeza* **),**
expl_retract(*Cabeza* **),**
validar_porque(*Cola* **));**

(no(*Cabeza_sin_no* **),**
explicacion_no(*Cabeza_sin_no* **),**
expl_retract(*Cabeza_sin_no* **)**
atom_concat(*Cabeza_sin_no, 'no', Cabeza* **),**
validar_porque(*Cola* **)).**

validar_porque(*[ ]* **).**

expl_retract(*Hecho* **) :- (retract(si(** *Hecho* **)), retract(explicacion_si(** *Hecho* **)));**
(retract(no(*Hecho* **)), retract(explicacion_no(** *Hecho* **))).**

Como se puede apreciar en el código, estamos haciendo uso de una función recursiva que comprueba primero si se ha asignado **si**(*Hecho*) y después si existe una **explicacion**(*Hecho*). A continuación llama a una función que se encarga de borrar ambos y finalmente vuelve a llamarse a sí misma. En caso de no encontrar confirmaciones positivas, pasa a las negativas, siendo la única diferencia una concatenación de un no por delante del hecho, para que la explicación tenga sentido. Estos elementos se irán agregando a la lista hasta que no se encuentren más, donde entrará en juego el caso base, asignando a lo que sería la cola "final" de la lista la lista vacía ([]), y cerrando así la lista.

Y esto, junto con unas menores modificaciones, sería, ahora sí, el final del código. Este puede verse completo en la página 38.

5. NIVEL DE AMBICIÓN ESCOGIDO

Consideramos que el nivel "millorat" de ambición es el más adecuado para garantizar un buen diseño y funcionamiento del sistema, en comparación con la complejidad de implementar las características adicionales del nivel max. Además, priorizamos garantizar un informe de alta calidad para que todo el trabajo sea comprensible y transparente, en lugar de enfocarnos en alcanzar un nivel de ambición alto pero con un informe de menor calidad.

En nuestro trabajo hemos desarrollado un sistema avanzado de diagnóstico para problemas específicos. El sistema es interactivo y se comunica con los usuarios a través de preguntas para determinar la presencia de ciertos síntomas. Una vez establecida la presencia de síntomas, el sistema busca establecer la causa de los mismos. Si no es posible establecer una causa, se informa al usuario y la sesión actual finaliza. Sin embargo, el usuario tiene la opción de iniciar una nueva sesión hasta que desee finalizar las consultas. Además, se ha incorporado la posibilidad de que el usuario pueda solicitar una explicación detallada de cómo se llegó al diagnóstico, con el objetivo de proporcionar un mayor nivel de comprensión y transparencia del proceso.

6. MANUAL

6. 1. INSTALACIÓN DEL PROGRAMA

- Lo primero que se ha de hacer es descargar el archivo al que dejaremos un enlace en la sección del código. Compruebe que el archivo es una extenión .pl (legible por prolog)

- A continuación, abra el programa SWI-prolog, o cualquier otro que sea capaz de ejecutar comandos de prolog.

- Compile en la aplicación el programa que se ha descargado. En SWI-Prolog esto se realiza haciendo click en: FILE → CONSULT → <seleccionar el archivo>.

- El programa estaría ya preparado para comenzar con el diagnóstico.

6. 2. EJECUCIÓN DEL PROGRAMA

- Para comenzar a utilizar nuestro programa de diagnóstico, deberá introducir:

  “ **inicio.** ”

  (*Recuerde que si no escribe un punto después de todo aquello que quiera que el prolog ejecute, este no lo considerará, pues aún considera que no se ha finalizado la línea*)

- Después, el programa le comenzará a hacer preguntas, responda:

  “si.” o **“s.”**
  en caso de que la respuesta a la pregunta sea afirmativa, y en caso contrario, responda:
  “no.” o **“n.”**

- De un momento a otro, el programa sacará una conclusión, y se la escribirá por pantalla, luego le preguntará si quiere saber por qué, a lo que usted responderá “**si.**” o “**s.**” en caso afirmativo, y cualquier otra palabra en caso negativo.

- Finalmente, el programa le preguntará si quiere continuar haciendo diagnósticos, a lo que tendrá que responder: “**si.**” o “**s.**” en caso afirmativo y cualquier otra palabra en caso negativo.

7. REFERENCIAS

[1] Author Info, How to Troubleshoot a PS2. Wikihow: [en línia], 2020. Disponible en: <https://www.wikihow.com/Troubleshoot-a-PS2>.

[2] Nicholas Congleton, How to Fix a PS2 Disc Read Error. Lifewire: [en línia], 2019. Disponible en: <https://www.lifewire.com/fix-ps2-disc-read-error-4773656>.

[3] Punkrules, Make PS2 Purple Discs Read in Your PS2 Simply by Using Tape. Autodesk Instructables: [en línia], 2008. Disponible en: <https://www.instructables.com/Make-PS2-Purple-Discs-Read-In-Your-PS2-Simply-By-U/>.

[4] SWI-Prolog: 1987. Disponible en: <https://www.swi-prolog.org/>.

[5] Chat OpenAI - Chat GPT. OpenAI: [en línia]. Disponible en: <https://chat.openai.com/chat>.

[6] Diagrams.net: [en línia]. Disponible en: <https://app.diagrams.net/>.

8. CÓDIGO

```
/*
  DIAGNÓSTICO DE UN FALLO EN LA CONSOLA: PlayStation 2

  -- Artur Aubach Altes & Pablo Barrenechea Perea --
*/

    inicio :- deshacer,
    nl,nl,
    hipotesis(Hipotesis_confirmada),
    write('Por favor: '),
    write(Hipotesis_confirmada),
    nl,nl,nl,

    write('Quieres saber por que? '),
    read(Confirmacion_porque),
    nl,
    ((Confirmacion_porque == si ; Confirmacion_porque == s)->
(explicacion(Hipotesis_confirmada), nl,nl); nl),
    nl,
    write('Quieres realizar un consulta nueva?'),
    read(Confirmacion_consulta),
    ((Confirmacion_consulta == si ;
      Confirmacion_consulta == s) ->
                          (nl,nl,nl,

write(q_____________________________________NUEVA_CONSULTA_____________________
_____________p),
                          nl,
                          inicio) ;

                              deshacer).


/* Hipótesis por confirmar */

%1  La consola no se enciende
hipotesis(asegurese_de_que_los_cables_estan_correctamente_conectados_a_
las_ranuras)    :- hipotesis_E_1, !.
hipotesis(intente_reparar_el_cable_o_compre_uno_nuevo)      :-
hipotesis_E_2, !.
```

```
hipotesis(encienda_el_boton_de_alimentacion_principal)    :-
hipotesis_E_3, !.
hipotesis(compre_otra_consola_o_intente_repararla_internamente) :-
verifica(se_enciende_la_consola,no,no), !.


%2  Los controladores no funcionan
hipotesis(asegurese_de_que_los_cables_del_controlador_estan_correctamen
te_conectados_a_las_ranuras)    :- hipotesis_C_1, !.
hipotesis(intente_reparar_el_cable_del_controlador_o_compre_uno_nuevo)
:- hipotesis_C_2, !.
hipotesis(limpie_con_cautela_las_ranuras_utilizando_un_trapo_envuelto_e
n_la_punta_del_dedo_indice)    :- hipotesis_C_3, !.
hipotesis(compre_otra_consola_o_intente_repararla_internamente) :-
verifica(funcionan_los_controladores,no,no), !.


%3  La bandeja del disco no se abre
hipotesis(compruebe_que_el_problema_estaria_solucionado)    :-
hipotesis_S_1, !.
hipotesis(compruebe_que_el_problema_estaria_solucionado)      :-
hipotesis_S_2, !.
hipotesis(compre_otra_consola_o_intente_repararla_internamente)    :-
hipotesis_S_3, !.
hipotesis(fuera_del_dominio):-
verifica(se_abre_la_bandeja_del_juego,no,no), !.


%4  Un disco de un juego no funciona
hipotesis(compre_un_disco_nuevo)   :- hipotesis_J_1_1, !.
hipotesis(coloque_un_trozo_de_una_pulgada_de_cinta_adhesiva_a_cada_lado
_del_disco_muy_cerca_del_horificio_central) :- hipotesis_J_1_2_1, !.
hipotesis(compruebe_que_el_problema_estaria_solucionado)   :-
hipotesis_J_1_2_2, !.
hipotesis(limpie_con_delicadeza_el_laser_optico_utilizando_alcohol) :-
hipotesis_J_1_2_3, !.
hipotesis(limpie_con_cuidado_el_disco) :- hipotesis_J_2_1, !.
hipotesis(pruebe_a_arreglar_el_disco_con_un_kit_de_reparacion) :-
hipotesis_J_2_2, !.
```

```
hipotesis(fuera_del_dominio).



%1  La consola no se enciende
hipotesis_E_1 :- (verifica(se_enciende_la_consola,no,no),

verifica(estan_correctamente_conectados_todos_los_cables,si,no)).

hipotesis_E_2 :- (verifica(se_enciende_la_consola,no,no),
          verifica(algun_cable_esta_roto,si,no)).

hipotesis_E_3 :- verifica(se_enciende_la_consola,no,no),

verifica(esta_encendido_el_boton_de_alimentacion_principal,no,no).


%2  Los controladores no funcionan
hipotesis_C_1 :- verifica(funcionan_los_controladores,no,no),

verifica(estan_correctamente_conectados_todos_los_cables_de_los_control
adores,no,no).

hipotesis_C_2 :- verifica(funcionan_los_controladores,no,no),
          verifica(el_cable_del_controlador_esta_roto,si,no).

hipotesis_C_3 :- verifica(funcionan_los_controladores,no,no),
          verifica(las_ranuras_estan_obstruidas,si,no).


%3  La bandeja del disco no se abre
hipotesis_S_1 :- verifica(se_abre_la_bandeja_del_juego,no,no),
          verifica(reiniciando_la_consola_se_abre_la_bandeja,si,no).

hipotesis_S_2 :- verifica(se_abre_la_bandeja_del_juego,no,no),
          verifica(moviendo_la_bandeja_con_el_dedo_se_abre,si,no).

hipotesis_S_3 :- verifica(se_abre_la_bandeja_del_juego,no,no),

verifica(se_escucha_un_ruido_raro_si_le_quitas_la_tapa_a_la_consola,si,
no).
```

```
%4  Un disco de un juego no funciona
hipotesis_J_1_1 :- verifica(funciona_el_disco_de_un_juego, no, si),
        verifica(esta_en_mal_estado_el_disco, no, si),
        verifica(funciona_con_otros_discos, si, si),
        verifica(siguen_algun_patron_los_discos,no, si).

hipotesis_J_1_2_1 :- verifica(funciona_el_disco_de_un_juego, no, si),
          verifica(esta_en_mal_estado_el_disco, no, si),
          verifica(funciona_con_otros_discos, si, si),
          verifica(siguen_algun_patron_los_discos, si, si),
          verifica(son_del_mismo_color_los_discos, si, si),
          verifica(son_de_color_azul_o_morado_los_discos, si, si).

hipotesis_J_1_2_2 :- verifica(funciona_el_disco_de_un_juego, no, si),
          verifica(esta_en_mal_estado_el_disco, no, si),
          verifica(funciona_con_otros_discos, si, si),
          verifica(siguen_algun_patron_los_discos,si, si),
          verifica(son_del_mismo_color_los_discos, si, si),
          verifica(son_de_color_azul_o_morado_los_discos, no, si),
          verifica(funciona_despues_de_poner_un_disco_de_limpieza, si,
si).

hipotesis_J_1_2_3 :-  verifica(funciona_el_disco_de_un_juego, no, si),
          verifica(esta_en_mal_estado_el_disco, no, si),
          verifica(funciona_con_otros_discos, si, si),
          verifica(siguen_algun_patron_los_discos,si, si),
          verifica(son_del_mismo_color_los_discos, si, si),
          verifica(son_de_color_azul_o_morado_los_discos, no, si),
          verifica(funciona_despues_de_poner_un_disco_de_limpieza, no,
si).

hipotesis_J_2_1 :- verifica(funciona_el_disco_de_un_juego, no, si),
        verifica(esta_en_mal_estado_el_disco, si, si),
        verifica(esta_sucio_el_disco, si, si).

hipotesis_J_2_2 :- verifica(funciona_el_disco_de_un_juego, no, si),
        verifica(esta_en_mal_estado_el_disco, si, si),
        verifica(esta_rayado_el_disco, si, si).
```

```
% Confirmación de un hecho
verifica(Hecho, Confirmar_hecho, Doble_explicacion) :-
          ((Confirmar_hecho == no) ,
          (no(Hecho) ->
              true ;
              (si(Hecho)) ->
                  fail ;
                  pregunta(Hecho, Confirmar_hecho, Doble_explicacion)))
;

          ((Confirmar_hecho == si) ,
            (si(Hecho) ->
                true ;
                (no(Hecho) ->
                    fail ;
                    pregunta(Hecho,Confirmar_hecho,
Doble_explicacion)))).



% Preguntar al usuario
pregunta(Pregunta, Confirmar_hecho, Doble_explicacion) :-
  write(Pregunta),
  write('? '),
  read(Respuesta),
  nl,
  (( (Confirmar_hecho == si) ,
      ((Respuesta == si ; Respuesta == s)
        ->  (assert(si(Pregunta)), assert(explicacion_si(Pregunta))) ;

      (assert(no(Pregunta)),
                (Doble_explicacion == si ->
          (assert(explicacion_no(Pregunta)));

          fail)),
                fail)) ;

   ( (Confirmar_hecho == no) ,
      ((Respuesta == no ; Respuesta == n)
        ->  (assert(no(Pregunta)), assert(explicacion_no(Pregunta))) ;
            (assert(si(Pregunta)),
                (Doble_explicacion == si ->
(assert(explicacion_si(Pregunta))); fail)),
```

```
                fail))).


% EStructurar la explicacion
explicacion(Hipotesis_confirmada) :-
  write(Hipotesis_confirmada),
  write(', porque: '),
  Hipotesis_confirmada \= fuera_del_dominio
    -> (validar_porque(Llista), write(Llista));

  write('no existen suficientes bases en nuestro conocimiento del
dominio como para diagnosticar su problema.').


validar_porque([Cabeza_lista|Cola_lista]) :- (si(Cabeza_lista),

explicacion_si(Cabeza_lista),
                                          expl_retract(Cabeza_lista),
                                          validar_porque(Cola_lista))
;
                                          (no(Cabeza_lista_sin_no),

explicacion_no(Cabeza_lista_sin_no),

expl_retract(Cabeza_lista_sin_no),
atom_concat('no_',Cabeza_lista_sin_no,Cabeza_lista),
                                          validar_porque(Cola_lista)).
validar_porque([]).


expl_retract(Hecho) :- (retract(si(Hecho)),
                    retract(explicacion_si(Hecho)));

           (retract(no(Hecho)),
                    retract(explicacion_no(Hecho))).



% Declaracion dinamica de todas aquellas funciones que vayamos a
modificar durante el programa
:- dynamic si/1, no/1, explicacion_si/1, explicacion_no/1.
```

```
% Deshacer todo aquello que haya podido quedar antes de iniciar y al
acabar el programa
deshacer :- retract(si(_)), fail.
deshacer :- retract(no(_)), fail.
deshacer :- retract(explicacion_si(_)), fail.
deshacer :- retract(explicacion_no(_)), fail.
deshacer.
```