Universidad Complutense de Madrid

Facultad de Informática

Proyecto de Sistemas Informáticos

Generación automática de casos de prueba para consultas SQL en el sistema DES

Ángel Panizo LLedot
Cristina García Sanz
Irene Núñez de Arenas Besteiro

Directora: Yolanda García Ruiz

Junio 2013

Agradecimientos

A Yolanda, por dirigirnos y ayudarnos y sobre todo por prestarnos sus estudios e investigaciones como base para nuestro proyecto.

A nuestos familiares, parejas y amigos por soportar nuestros agobios y nerviosismos.

Y a todos aquellos profesores y compañeros que nos han acompañado a lo largo de nuestro periplo universitario, porque gracias a sus conocimientos y enseñanzas ha sido posible sentar las bases para la realización de este proyecto de fin de carrera.

Autorización

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

Firma de los autores:

Angel Panizo Lledot

Cristina García Sanz

Irene Núñez de Arenas Besteiro

Índice

Índice de Figuras

Resumen

El sistema DES v3.0 (Datalog Educational System) es un gestor de bases de datos deductivas y relacionales. Posee módulos que permiten la definición de datos y su manipulación mediante el lenguaje SQL. Además, con respecto a las consultas escritas en SQL, posee un generador automático de casos de prueba para vistas, con el fin de capturar posibles errores de diseño en sus consultas.

Este proyecto ha sido desarrollado con el propósito de extender la generación automática de casos de prueba del sistema DES. El funcionamiento es el siguiente: Dado el nombre de una vista a probar, definida en el esquema de la base de datos a partir de otras vistas y tablas, nuestra herramienta crea primero las fórmulas que contienen las condiciones que una instancia de la base de datos tiene que cumplir para que la consulta de la vista devuelva al menos un registro (lo que se considera como un caso de prueba positivo), y después traduce esas fórmulas a lenguaje de restricciones, cuya solución constituye el caso de prueba deseado.

Esta separación entre generación de fórmulas y de restricciones hace que el sistema se vuelva más modular y aporta bastantes ventajas, como la posibilidad de modificar un módulo sin que afecte a otras partes del código o la facilidad de detección y aislamiento de errores.

Palabras clave

Base de datos, Caso de prueba, DES, Prolog, Restricciones, SQL

Abstract

The DES system v3.0 (Datalog Educational System) is a deductive and relational database manager. It has modules for data definition and manipulation using SQL language. Besides, regarding the queries written in SQL, implements an automatic test case generator for SQL views in order to catch design errors in their queries.

This project has been developed with the purpose of extending the automatic test cases generation for DES system. It works as follows: Given the name of a view to be tested, defined in the schema of the database in terms of other views and schema tables, our tool first creates the formulas that contains the conditions that a database instance has to satisfy so that the view query returns at least one record (which is considered to be a positive test case), and then translates these formulas into constraint language, whose solution constitute the desired test case.

This separation between formulas and constraints generation makes the system become more modular and provides several advantages, such as the possibility to modify a module without affecting other parts of the code or an easier way to detect and isolate errors.

Keywords

Data Base, Test Case, DES, Prolog, Constraints, SQL

Capítulo 1

Introducción

1.1 Explicación del problema

> *La información es un conjunto organizado de datos procesados, que constituyen un mensaje que cambia el estado de conocimiento del sujeto o sistema que recibe dicho mensaje [9].*

Vivimos en la era de la información, información que se genera, se almacena, se recupera y se transmite. Inventos como la imprenta, la radio, el teléfono, la televisión o internet favorecen el intercambio de todo tipo de datos y contenidos a nivel global, provocando una verdadera avalancha de información.

De esta necesidad de información, surge la necesidad de almacenamiento persistente. Es posible almacenar datos en múltiples soportes y de diferentes formas, pero el número de datos que es necesario conservar crece de manera imparable haciendo cada vez más extenso y complejo el conjunto de los mismos.

> *Una base de datos es un conjunto de datos pertenecientes a un mismo contexto y almacenados sistemáticamente para su posterior uso [8].*

Actualmente, y debido al desarrollo tecnológico de campos como la informática y la electrónica, la mayoría de las bases de datos están en formato digital ofreciendo así un amplio rango de soluciones al problema del almacenamiento de datos. Igualmente, existen programas denominados sistemas gestores de bases de datos capaces de manejar estas grandes colecciones de datos de forma rápida y estructurada.

Existen diferentes técnicas para modelar bases de datos, siendo el modelo relacional uno de los mas utilizados en la actualidad. El modelo relacional es un modelo de datos basado en la lógica de predicados y en la teoría de conjuntos. Las bases de datos relacionales tienen las siguientes características:

- Los datos y las relaciones entre ellos se almacenan en *relaciones*.
- Cada relación se puede ver como una tabla.
- Cada tabla es, a su vez, un conjunto de registros o tuplas.

- Cada registro es un conjunto de campos o atributos (columnas) que caracterizan o describen al registro.
- Cada atributo tiene asociado un dominio, es decir, un conjunto de valores permitidos.
- Los registros de las tablas se identifican de forma unívoca mediante la clave primaria.
- Las claves ajenas identifican una columna o grupo de columnas en una tabla (tabla hija) que se refieren a una columna o grupo de columnas en otra tabla (tabla referenciada). Las columnas en la tabla hija deben ser la clave primaria u otra clave candidata en la tabla referenciada.

Como se desprende de lo anteriormente indicado, hoy en día el uso de las bases de datos está ligado prácticamente a cualquier tipo de aplicación informática, para la gestión de empresas e instituciones públicas, para la gestión de historiales médicos en la sanidad, en entornos científicos para almacenar la información experimental o simplemente en casa para llevar la cuenta de las películas en DVD que se poseen.

Una de las tareas más importantes a las que se enfrenta el creador de una aplicación informática es el diseño de la base de datos en la que se almacenarán los datos necesarios para el buen funcionamiento de la misma de forma organizada y haciendo posible su acceso de manera eficiente. Aún siendo tan importante, esta fase del proceso de creación no cuenta con mucha ayuda a la hora de depurar y probar su funcionamiento, ya que el uso de herramientas complementarias para tal efecto parece no estar muy extendido. Comprobar la corrección de una consulta es difícil utilizando instancias reales, ya sea porque el número de filas en las tablas suele ser muy elevado en bases de datos pobladas o porque haya pocos datos en la fase inicial de desarrollo. Por eso, es importante disponer de instancias válidas de la base de datos que cumplan con ciertos criterios de calidad y que constituyen lo que se conoce como *casos de prueba*. Además, es conveniente que estas instancias sean de pequeño tamaño y que permitan detectar el mayor número de errores en las consultas.

Para intentar simplificar en cierto modo la depuración en el mundo de las bases de datos relacionales, hemos desarrollado una herramienta que permite encontrar una instancia de la base de datos que cumple los criterios de calidad especificados por el criterio de medida de predicado. Como se explicará en capítulos sucesivos, el sistema desarrollado recibe como entrada una serie de sentencias SQL que definen el esquema de la base de datos y el nombre de la vista para la que se quiere obtener el caso de prueba. Estos datos de entrada son analizados para reducir el problema de obtener el caso de prueba a un problema de resolución de restricciones y, por último, el sistema devuelve un caso de prueba para la vista indicada, que cumple todas las restricciones impuestas por la base de datos.

1.2 Conceptos previos

1.2.1 Funcionamiento de la programación basada en restricciones

La programación con restricciones presenta diferencias con respecto a la programación tradicional, en la cual se establecen una serie de operaciones para resolver un problema.

Los pasos típicos a seguir en cualquier lenguaje de programación con restricciones se detallan a continuación:

1. Inicialmente se genera un número de variables sin instanciar; es decir, sin valor asignado, necesarias para resolver el problema. Estas variables de denominan variables simbólicas. Para diferenciar unas variables de otras, se suele asignar a cada una de ellas un nombre distinto. Además, se debe definir el dominio o rango de valores que éstas pueden tomar.

2. Posteriormente, se fijan una serie de restricciones que expresan las relaciones entre las variables, limitando así la combinación de valores que se pueden asignar a cada una de ellas. Toda solución al problema debe satisfacer estas restricciones.

3. Una vez fijadas dichas restricciones, se prueba a realizar un etiquetado o labeling a las variables, que consiste en intentar encontrar un valor (perteneciente al dominio finito fijado en el paso 1) para cada una de ellas, de forma que se consiga satisfacer todas las restricciones del problema de manera simultánea; o bien, por el contrario, no se logren satisfacer, en cuyo caso el etiquetado falla por lo que las variables quedan sin instanciar.

Considérese el siguiente ejemplo:

En el puzzle "Send More Money" [5] las variables son las letras S, E, N, D, M, O, R, Y. Cada letra representa un dígito entre 0 y 9. El problema consiste en asignar un valor a cada letra, de forma que SEND + MORE = MONEY.

Paso 1: Se generan las variables S, E, N, D, M, O, R, Y. El dominio de este puzzle se establece mediante el predicado `domain/3`, y requiere que cada variable toma valor entre 0 y 9 y que S y M sean mayores que cero.

Paso 2: Las dos restricciones de este problema son que todas las letras tomen distintos valores (predicado `all_different/1`) y que se cumpla la ecuación SEND + MORE = MONEY (predicado `sum/8`).

Paso 3: Finalmente, se realiza el etiquetado a las variables mediante el predicado `labeling/2` utilizando la estrategia de vuelta atrás. Se pueden seleccionar diferentes estrategias de búsqueda en el parámetro *Type* de este predicado, en este ejemplo se usa la estrategia de búsqueda por defecto, probar valores en orden ascendente.

```
mm([S,E,N,D,M,O,R,Y], Type) :-
      % Paso 1
      domain([S,E,N,D,M,O,R,Y], 0, 9),
      S#>0, M#>0,
      % Paso 2
      all_different([S,E,N,D,M,O,R,Y]),
      sum(S,E,N,D,M,O,R,Y),
      % Paso 3
      labeling(Type, [S,E,N,D,M,O,R,Y]).

sum(S, E, N, D, M, O, R, Y) :-
      1000*S + 100*E + 10*N + D + 1000*M + 100*O + 10*R + E
      #= 10000*M + 1000*O + 100*N + 10*E + Y.
```

Llamada al procedimiento mm:

```
|?- mm([S,E,N,D,M,O,R,Y], []).
```

Tras ejecutar, se asigna los siguientes valores a cada una de las variables:

D = 7, E = 5, M = 1, N = 6, O = 0, R = 8, S = 9, Y = 2

1.2.2 Casos de prueba software

Las aplicaciones software se han convertido en una herramienta imprescindible, tanto para las empresas (bancos, supermercados, restaurantes, etc) como para los productos y servicios utilizados a diario por la población (teléfonos móviles, aviones, coches, etc)

Es inevitable que existan defectos en el software, dado que pueden darse condiciones ambientales que lo deterioren, y además, hay que tener en cuenta que el software es desarrollado por personas, y las personas son falibles.

Si un software falla, significa que no funciona como es esperado. Esto puede tener consecuencias desastrosas, tales como la pérdida de tiempo, la pérdida de dinero e incluso puede producir daños físicos a las personas. Por tanto, es imprescindible realizar pruebas, con el fin de detectar y solucionar errores en el código y así evitar las consecuencias negativas.

Un caso de prueba software se define como una serie de variables de entrada y condiciones de ejecución bajo las cuales se determina si un requisito de una aplicación es satisfactorio, comparando el resultado obtenido con el esperado. Se pueden realizar numerosos casos de prueba para determinar si un requisito es satisfactorio, pero es obligatorio que exista al menos uno para cada requisito.

El proceso de prueba aplicado a software es más exhaustivo y menos automático que, por ejemplo, el aplicado a una consulta SQL, ya que en la prueba de la implementación de una aplicación software existe el proceso manual por parte de los desarrolladores (o de un grupo especifico destinado a tal efecto, los testers) de buscar qué casos pueden fallar.

Hay que tener en cuenta que un caso de prueba debe ayudar a detectar fallos en un sistema. No obstante, si no se encuentran fallos, no se puede asegurar que el software no los tenga.

1.2.3 Casos de prueba para consultas SQL

Generar un caso de prueba para una consulta SQL no es una tarea sencilla. Hay muchos factores a tener en cuenta, tales como el dominio de entrada, o el estado en el que se encuentra la base de datos (restricciones, dependencias entre relaciones, etc) ya que cualquier cambio en su estructura puede cambiar el comportamiento de las consultas. Además, es difícil verificar los resultados de las pruebas dado que la semántica de SQL es compleja.

Existen numerosos procesos de prueba software. Uno de ellos es el denominado *white-box testing* o prueba de caja blanca. Este tipo de prueba se encarga de probar las funciones internas de un programa y consiste en ejecutar el programa en estudio eligiendo unos valores de entrada determinados, y comprobar si los resultados obtenidos en cada uno de los posibles flujos de ejecución se ajustan a las especificaciones del programa.

Coverage Rules

ID	Cat	Type	Subt	Loc	SQL	Text
0	S	B	B+F	w.1.1.[WHERE t.b > 1]	SELECT * FROM t WHERE (t.b = 2)	--Some row in the table such that: --The WHERE condition fulfills: --(B+) t.b = 2
1	S	B	B=F	w.1.1.[WHERE t.b > 1]	SELECT * FROM t WHERE (t.b = 1)	--Some row in the table such that: --The WHERE condition fulfills: --(B=) t.b = 1
2	S	B	B-F	w.1.1.[WHERE t.b > 1]	SELECT * FROM t WHERE (t.b = 0)	--Some row in the table such that: --The WHERE condition fulfills: --(B-) t.b = 0
3	S	N	NF	w.1.1.[t.b]	SELECT * FROM t WHERE (t.b IS NULL)	--Some row in the table such that: --The WHERE condition fulfills: --(N) t.b is NULL

Figura 1.1: Ejemplo de Criterio de Cobertura

Las pruebas de caja blanca son adecuadas para las consultas SQL dado que, para comprobar si una consulta es correcta, es importante generar un conjunto de casos de prueba que cubra el mayor número de situaciones posibles.

El problema de encontrar un conjunto completo de casos de prueba, que cubra todas y cada una de las situaciones posibles que pueden darse en un programa, es generalmente indecidible. No obstante, se han definido distintos criterios de recubrimiento para generar casos de prueba que sean completos en al menos alguna propiedad deseada [4]. Uno de ellos es el criterio de recubrimiento SQLFpc (SQL Full Predicate Coverage). Este criterio permite medir la calidad del conjunto total de casos de prueba y viene definido mediante un conjunto de reglas derivadas de la semántica de la consulta. Cada una de estas reglas tiene una representación mediante una consulta escrita en lenguaje SQL. De esta forma, un conjunto de casos de prueba cumplen el criterio SQLFpc si se verifica que todas las consultas asociadas a las reglas devuelven, al menos, una tupla como resultado.

Por ejemplo, considérese una tabla t con dos campos de tipo entero: a y b, y considérese también la siguiente consulta:

```
SELECT *
FROM t
WHERE t.b > 1;
```

Las reglas definidas por este criterio de cobertura, y su representación como consultas SQL, para la consulta dada se muestran en la Figura 1.1 [7].

1.3 Características de nuestro proyecto

El sistema DES [6] permite gestionar tanto bases de datos deductivas, como bases de datos relacionales. Con respecto a las bases de datos relacionales, DES tiene un generador de casos de prueba individuales para una vista dada de forma que dicho caso de prueba cumple un criterio de cobertura de predicado. Es lo que en [6] denominan PNTC (Positive and Negative Test Case).

Nuestro proyecto se centra en la generación de otro tipo de casos de prueba, son los llamados casos de prueba positivos (PTC en [6]).

Una instancia de la base de datos d es un caso de prueba positivo para una vista v si el resultado de v es no vacío. En una consulta básica, un caso de prueba positivo busca un registro en el dominio de la consulta que satisfaga la condición de la cláusula `WHERE`. En el caso de una consulta agrupada, deberá existir un grupo válido que cumpla la condición de la cláusula `HAVING` (lo que a su vez implica que todas sus filas cumplan la condición de la cláusula `WHERE`). Si se trata de la unión o intersección de dos consultas, de la forma C_1 `UNION/INTERSECT` C_2 , la instancia d generada será un caso de prueba positivo para v si lo es para C_1 o C_2 en el caso de la unión, o bien lo es para C_1 y C_2 en el caso de la intersección. [4]

Por tanto, nuestra aplicación no crea por sí misma un conjunto completo de casos de prueba que cumpla el criterio SQLFpc. No obstante, una vez generadas las consultas que representan cada una de las reglas que define el criterio, podrá utilizarse para generar los casos de prueba individuales sobre cada una de las mismas, que conformarán el conjunto de casos de prueba final, dotando así al sistema DES de herramientas suficientes para probar de consultas.

Capítulo 2

Diseño

2.1 Estado del arte

Comprobar la corrección de un programa software es una fase crucial en el ciclo de vida de una aplicación. No es raro que las pruebas de detección y corrección de errores se lleven gran parte del coste de un proyecto ya que es una labor compleja que conlleva muchas horas de trabajo. En los comienzos de la ingeniería del software muchas de estas comprobaciones se realizaban a mano. Sin embargo, la tendencia actual es automatizar estas pruebas de tal forma que se conviertan en programas en sí mismas.

Se puede decir que la fase de pruebas en una aplicación cualquiera que interactúa con una base de datos a través de su API se puede reducir, en gran medida, a hacer pruebas sobre dicha base de datos. Por lo tanto generar instancias de la base de datos que puedan mostrar posibles errores durante las pruebas del programa se ha convertido en una tarea importante.

Estudios recientes [2] exploran estas ideas en diferentes formas:

- **RQP (Reverse Query Processing):** Dados un esquema de la base de datos, una consulta SQL y el resultado para dicha consulta, se trata de obtener una posible instancia de la base de datos que tiene la propiedad de que, si se le aplica la consulta proporcionada como entrada, se obtendrá el resultado también proporcionado.

  Con el objeto de generar datos que satisfagan las restricciones del esquema de la base de datos así como de la consulta, ésta es analizada sintácticamente y tratada en base al álgebra relacional inversa, proceso que generalmente implica la generación de datos. Por ejemplo: el operador inverso de la proyección genera columnas, mientras que el operador de proyección hacia delante elimina columnas.

  Dado que son muchas las bases de datos que pueden ser generadas a partir de una consulta y un resultado para dicha consulta, es necesario que la instancia generada sea lo más pequeña posible para mayor rapidez en las pruebas funcionales de una aplicación.

  Una posible implementación de estas ideas se puede ver en [1].

- **SQP (Symbolic Query Processing):** Dados un esquema de la base de datos, una serie de restricciones en los operadores de una consulta (por ejemplo, restricciones de cardinalidad) y una consulta paramétrica, se trata de obtener una instancia de la base de datos y valores para los parámetros de la consulta.

  El proceso de generación de datos involucra una base de datos simbólica, que consiste en múltiples tablas simbólicas en las que los datos son representados mediante símbolos en vez de valores concretos, y una tabla especial que contiene las restricciones definidas por el usuario en forma de tuplas. Una vez creada esta base de datos simbólica se usa un resolutor para instanciar las restricciones de las tuplas y obtener así un caso de prueba.

  Una posible implementación de estas ideas (QAGen) se puede ver en [3].

2.1.1 Aplicando Programación Lógica con Restricciones

En el artículo que da nombre a este subapartado [4] se exploran las ideas del procesamiento simbólico de consultas para, dado un conjunto de vistas relacionadas, transformarlo en un problema de resolución de restricciones cuya solución proporcione una instancia de la base de datos, que constituye un caso de prueba.

El problema de generar casos de prueba para vistas relacionadas no puede reducirse a resolver el problema para cada una por separado:

```
CREATE OR REPLACE VIEW v1(b) AS
SELECT t.a
FROM t
WHERE t.a > 8

CREATE OR REPLACE VIEW v2(c) AS
SELECT v1.b
FROM v1
WHERE v1.b > 5
```

Sean v1 y v2 dos vistas que suponen la existencia de una tabla t con un solo atributo a, entero, un caso de prueba positivo para v2 sin considerar la relación entre ambas vistas podría ser una fila para v1 que tuviera v1.b=6 ya que 6>5, que es la condición que aparece en v2. Sin embargo, 6 no es un valor posible para v1 puesto que v1.b sólo puede contener valores mayores que 8, lo que demuestra que la relación entre las dos vistas se debe tener en cuenta.

Llegados a este punto, es importante contemplar el concepto de *vista*. Las vistas pueden ser consideradas como nuevas tablas creadas dinámicamente a partir de tablas ya existentes usando una consulta y permitiendo el renombramiento de atributos. La consulta generadora de una vista puede depender de las tablas del esquema de la base de datos (en el ejemplo anterior v1) o también de otras vistas definidas con anterioridad (v2 en el ejemplo).

Es posible generar el árbol de dependencias de una vista. En la Figura 2.1 se muestra como ejemplo el árbol de dependencias de la vista v2. Se trata de un árbol en cuya raíz aparece la vista objeto de estudio y cuyos hijos son los árboles de dependencia de las relaciones que aparecen en la cláusula *from* de la consulta que la define.

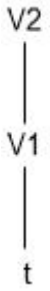

Figura 2.1: Árbol de dependencias del ejemplo

Aplicando Programación Lógica con Restricciones es posible generar de forma automática casos de prueba para vistas SQL. El proceso a seguir podría resumirse como sigue:

1. Crear una instancia simbólica de la base de datos. Cada tabla contendrá un número arbitrario de filas, y el valor de cada atributo en cada fila corresponderá a una variable lógica con sus restricciones de integridad asociadas.

2. Definir el conjunto de restricciones que se deben cumplir. Se trata por un lado de las restricciones asociadas a la integridad del esquema de la base de datos, como son las claves primarias y ajenas, y por otro lado las restricciones derivadas de la semántica de la vista.

3. Obtener un caso de prueba como solución al problema de satisfacer las restricciones.

Existe una implementación de estas ideas, como un componente de DES, que tomaremos como base de nuestro proyecto. En esta implementación la entrada consiste en un esquema de la base de datos D, definida como un conjunto de tablas τ, restricciones ς y vistas ν, así como las restricciones de integridad para los diferentes atributos, y una vista V. La salida será una instancia d de la base de datos D tal que d es un caso de prueba para la vista dada V.

2.1.2 DES: Datalog Educational System

El sistema DES, Datalog Educational System [6] permite gestionar tanto bases de datos deductivas como bases de datos relacionales. Está basado en Prolog y es de código abierto, portátil y multiplataforma. Ha sido creado como un sistema simple e interactivo para estudiantes, con el objeto de ilustrar los conceptos básicos acerca de Datalog, Álgebra Relacional y SQL.

Este sistema está implementado en Prolog y tiene un analizador sintáctico para consultas y vistas SQL, así como un sistema de inferencia para vistas. Esto nos permite beneficiarnos de las facilidades que aporta DES para el tratamiento de SQL mientras exploramos las características para resolver restricciones que proporciona Prolog.

DES puede ser ejecutado desde cualquier intérprete Prolog en cada sistema operativo que lo soporte. Nosotros hemos escogido `SICStus Prolog` como plataforma para nuestro desarrollo.

2.2 Objetivos

Tomando como base las ideas de [4] y partiendo de su implementación, hemos desarrollado este proyecto como una extensión de la generación de casos de prueba para vistas SQL en el sentido de que, en vez de interpretar las restricciones según se van generando a partir de la entrada al programa como una consulta SQL, primero se produce una traducción a lenguaje de restricciones para, posteriormente, enviar todas las fórmulas generadas a un resolutor de restricciones y obtener la salida deseada, un caso de prueba positivo.

Aparte de conseguir una implementación diferente, para intentar buscar rapidez y eficiencia en la ejecución del programa, otro objetivo de este proyecto es extender la sintaxis permitida. Para ello trataremos:

- Las consultas compuestas, es decir, aquellas que combinan los resultados de dos o mas consultas mediante operaciones de conjunto tales como `UNION` o `INTERSECT`.
- El operador para calcular el producto cartesiano `INNER JOIN`, así como su anidación.
- La cláusula `DISTINCT` para eliminar las filas duplicadas en los resultados de una instrucción `SELECT`.

Otra de nuestras aportaciones a la hora de ampliar la sintaxis permitida es incorporar el tratamiento de los valores nulos, tanto en la definición del esquema de la base de datos como a la hora de tratar expresiones lógicas, por ejemplo, en la cláusula `WHERE`.

El fin último de nuestro proyecto es, dado un conjunto de consultas independientes que constituyen la representación SQL de las reglas definidas por el criterio de cobertura SQLFpc para una determinada vista, tomar este conjunto de consultas y ejecutar cada una de ellas en nuestra aplicación para así obtener un caso de prueba positivo para cada una. El conjunto de casos de prueba así obtenido conformará el conjunto de casos de prueba necesario para detectar posibles errores en la consulta SQL inicial. Será objetivo de futuras investigaciones generar el conjunto de reglas que constituyen dicho recubrimiento y unir ambos proyectos para conseguir una aplicación que sea capaz de comprobar por sí sola la corrección o no de vistas relacionadas.

2.3 Sintaxis Permitida

En esta sección se describen las características del lenguaje SQL que contempla la herramienta.

Lo primero que se necesita a la hora de generar cualquier caso de prueba es definir el esquema de la base de datos sobre la que se inferirá la instancia que constituye dicho caso de prueba. Para tal efecto usaremos las sentencias SQL para la creación de bases de datos, siendo objetivo del analizador sintáctico de DES su tratamiento.

Además, las tablas sobre las que posteriormente se realizarán las consultas podrán incorporar restricciones sobre los campos que se deseen, ya sea de clave primaria (`PRIMARY KEY`), de integridad referencial (`FOREIGN KEY`) o para especificar que la columna no acepta valores nulos (`NOT NULL`). Dichas restricciones se tendrán en cuenta a la hora de lanzar un caso de prueba para una vista y generar una instancia de base de datos válida para ella. Los campos de las tablas deberán ser de tipo entero y podrán tomar valores nulos.

Otra cosa imprescindible a la hora de generar un caso de prueba positivo sobre una vista, es definir dicha vista e introducirla en la base de datos. Una vista puede hacer referencia a tablas, o bien a otras vistas definidas previamente.

Las vistas se definen mediante consultas. Los tres tipos de consultas que se permiten introducir en el sistema son las siguientes:

- **Consultas básicas.**
  Son todas las consultas de la forma:

  `SELECT` e_1 E_1`,...,`e_n E_n
  `FROM` R_1 B_1`,...,`R_m B_m
  `WHERE` C_w`c`

  No contienen agrupados ni subconsultas.

- **Consultas agrupadas.**
  Son todas las consultas de la forma:

  `SELECT` e_1 E_1`,...,`e_n E_n
  `FROM` R_1 B_1`,...,`R_m B_m
  `WHERE` C_w
  `GROUP BY` A_1`,...,`A_n
  `HAVING` C_h

  Incluyen la cláusula `GROUP BY` y pueden incluir o no la cláusula `HAVING`. En ésta última pueden aparecer las funciones de agregación `SUM` y `COUNT` con expresiones aritmético-lógicas.

- **Unión e intersección de consultas básicas y agrupadas.**
  Son todas las consultas de la forma:

  C_1
  `UNION / INTERSECT`
  C_2

  Siendo C_1 y C_2 consultas.

Además, los tres tipos de consultas admitidas soportan las siguientes características:

- Uso de `INNER JOIN` de dos o más consultas.
- Uso del operador `DISTINCT` en la sección `SELECT`.
- Realizar un renombramiento tanto de las expresiones de la cláusula `SELECT` , como de las relaciones de la cláusula `FROM`.
- Expresiones aritméticas en la sección `SELECT`.
- Expresiones lógicas en la sección `WHERE`: and, or, not, =, <>, <, <=, >, >= y combinaciones de ellas (not and, not or, etc).

- Expresiones aritméticas en la sección `WHERE`: +, -, /, * y combinaciones de ellas.
- Combinaciones de expresiones lógicas y aritméticas en la sección `WHERE`.
- Expresiones con valores `NULL` del tipo `A = NULL` y `A <> NULL`, siendo A un campo de una relación (tabla o vista).

Asimismo, dejamos fuera de este proyecto las consultas recursivas, los tipos de datos cadenas (var, varchar, etc), las consultas existenciales del tipo `EXISTS, NOT EXISTS, IN, ALL` y `ANY` y las subconsultas en la sección `FROM`.

Sin embargo, esta última funcionalidad se puede realizar, ya que es posible lanzar una consulta equivalente admitida por el sistema sustituyendo las subconsultas del `FROM` por vistas, de la siguiente manera:

Supóngase que existe la siguiente tabla:

```
CREATE OR REPLACE TABLE r(a int PRIMARY KEY, b int)
```

Y la siguiente vista que presenta una consulta con una subconsulta en la sección `FROM`:

```
CREATE OR REPLACE VIEW w(e1,e2) AS
SELECT v.a1 as e1, v.a2 as e2
FROM (SELECT 2*a as a1, b+a as a2
      FROM r
      WHERE b+a > 0) v
WHERE v.a1 <= 1
```

Esta vista se puede reemplazar por dos vistas, una que contiene el resultado de la subconsulta del `FROM` y otra que corresponde a la consulta primera. Así, la vista anterior se podría reemplazar por las dos siguientes:

```
CREATE OR REPLACE VIEW v(a1,a2) AS
SELECT 2*a as a1, b+a as a2
FROM r
WHERE b+a > 0

CREATE OR REPLACE VIEW w(e1,e2) AS
SELECT v.a1 as e1, v.a2 as e2
FROM v as v1
WHERE v.a1 <= 1
```

Todas estas restricciones descritas podrán ser incorporadas a la aplicación en futuras versiones.

2.4 Generación de Fórmulas

En esta sección se describe el proceso por el cual a partir de una consulta SQL obtenemos un conjunto de formulas lógicas que posteriormente transformaremos en restricciones (véase subsección 1.2.1).

Esta es una de las partes principales de nuestro proyecto, en ella nos encargamos de generar una fórmula para cada una de las tuplas de la instacia de la base de datos. Para ello seguimos los métodos descritos en [4].

Dada una instancia d de la base de datos, para cada relación R se define el conjunto $\Phi(R)$ que recoge la relación entre fórmulas y tuplas. Una tupla t estará en la relación R si la fórmula asociada a t se cumple en la instancia de la base de datos d.

A continuación detallamos cómo obtenemos cada una de estas fórmulas.

Procesamiento de tablas

Para cada una de las tablas de la base de datos, construimos el conjunto $\Phi(R)$ de pares de la forma (f, t) donde f es una fórmula y t es una tupla. En este caso las fórmulas representan las restricciones de clave primaria(PK), clave ajena(FK) y restricciones de tipo `NOT NULL`.

- **Claves Primarias:** Cuando uno o varios campos de una tabla son denominados clave primaria, quiere decir que cualquier tupla de esa tabla es inequívocamente identificada a través de esos campos. En otras palabras, si uno o varios campos de una tabla son declarados clave primaria, en esa tabla no pueden haber valores repetidos para los mismos.

  Tomemos como ejemplo la tabla $T(PK(a, b), c)$ como la tabla T con tres campos a,b,c siendo clave primaria los dos primeros. Supongamos que la instancia de la tabla T contiene las siguientes tuplas: $[a_1, b_1, c_1], [a_2, b_2, c_2], [a_3, b_3, c_3]$. La fórmula de clave primaria para la tupla $[a_1, b_1, c_1]$ de T sería la siguiente:

  $$(((a_1 \neq a_2 \wedge a_1 \neq null \wedge a_2 \neq null) \vee (b_1 \neq b_2 \wedge b_1 \neq null \wedge b_2 \neq null)) \wedge$$

  $$((a_1 \neq a_3 \wedge a_1 \neq null \wedge a_3 \neq null) \vee (b_1 \neq b_3 \wedge b_1 \neq null \wedge b_3 \neq null)))$$

  Sea $T(A1, ..., Ar)$ una tabla del esquema de base de datos con clave primaria $Pk(T) = \{A1, ...Am/...\}$. Sea $d(T) = \{\mu_1, ..., \mu_n\}$ una instancia de la tabla T en d. Para cada tupla $\mu_i \in d(T)$ definimos la fórmula Φ_{1i}:

  $$\Phi_{1i} = \bigwedge_{j=1, j\neq i}^{n} (\bigvee_{k=1}^{m} (\mu_i(A_k) \neq \mu_j(A_k) \wedge \mu_i(A_k) \neq null \wedge \mu_j(A_k) \neq null))$$

  Para el caso en el que $PK(T) = \emptyset$, se tiene que $\Phi_{1i} = true \ \forall i = 1...n$.

- **Claves ajenas:** Cuando en una tabla uno o varios campos son declarados como clave ajena (`FOREIGN KEY`) quiere decir que hace referencia a otro de otra tabla, es decir, que tiene que haber un campo con el mismo valor en la tabla a la que hace referencia.

  Sea el siguiente ejemplo con dos tablas: $R(a, b)$ y $T(a, b)$. Sea $FK(T, R) = (a, b), (a, b)$ una clave ajena. Consideremos que las tablas tienen las siguientes tuplas: en T, $\{[ta_1, tb_1], [ta_2, tb_2]\}$ y en R, $\{[ra_1, rb_1], [ra_2, rb_2]\}$. Para que las restricciones de clave ajena se cumplan tiene que ocurrir que los valores de los campos a y b de la tabla T aparezcan en la tabla R como campos a y b. Por ejemplo, la fórmula de clave ajena para la tupla $[ta_1, tb_1]$ de T sería la siguiente:

  $$(((ta_1 = ra_1 \wedge ta_1 \neq null \wedge ra_1 \neq null) \wedge (tb_1 = rb_1 \wedge tb_1 \neq null \wedge rb_1 \neq null)) \vee$$

$$((ta_1 = ra_2 \wedge ta_1 \neq null \wedge ra_2 \neq null) \wedge (tb_1 = rb_2 \wedge tb_1 \neq null \wedge rb_2 \neq null)))$$

Sean dos tablas $T(A1, ..., Ar)$ y $R(B1, ..., Bn)$ definidas en el esquema de base de datos. Sean $d(T) = \{\mu_1, ..., \mu_n\}$ y $d(R) = \{\nu_1, ..., \nu_n\}$ instancias de las tablas T y R en d respectivamente. Sea $d(T) = \{\mu_1, ..., \mu_n\}$ una instancia de T con n tuplas μ_i y $d(R) = \{\nu_1, ..., \nu_n\}$ una instancia de R con n tuplas ν_i.

Para cada $FK(T, R) = \{(A_1...A_m), (B_1...B_m) | m \leq r\}$ generamos la siguiente fórmula:

$$\Phi_{2i} = \bigvee_{j=1}^{n} (\bigwedge_{k=1}^{m} (\mu_i(A_k) = \nu_j(B_k) \wedge \mu_i(A_k) \neq null \wedge \nu_j(B_k) \neq null))$$

Si una tabla no tiene definida FK, se tiene que $\Phi_{2i} = true \ \forall i = 1...n$.

- **No nulos:** Cuando en una tabla un atributo es declarado como no nulo (`NOT NULL`), quiere decir que obligatoriamente deberá tomar un valor entero de entre aquellos que pertenecen al dominio que se está tratando.

  Si consideramos una tabla cualquiera $T(A_1...A_r)$ y $d(T) = \{\mu_1, ..., \mu_n\}$ una instancia de T, definimos Φ_{3i} como:

  $$\Phi_{3i} = \begin{cases} \mu_i(A_i) \neq null \text{ si } A_i \text{ está declarado como } \texttt{NOT NULL} \\ true \text{ en otro caso} \end{cases}$$

  Con las fórmulas Φ_{1i}, Φ_{2i} y Φ_{3i}, podemos construir el conjunto fórmula-tupla para la tabla T de la siguiente manera:

  $$\Phi(T) = \{((\Phi_{1i} \wedge \Phi_{2i} \wedge \Phi_{3i}), \mu_i) | \mu_i \in d(T), i = 1, ..., n\}$$

Procesamiento de vistas

Una vez obtenidas las fórmulas para las tablas pasamos a tratar las vistas. Las vistas, a diferencia de las tablas, se construyen sobre una consulta SQL, que a su vez usa otras vistas y tablas, por lo tanto la fórmula de una vista estará formada por las fórmulas de las vistas y de las tablas que use y las restricciones que ella misma genere.

Ahora veremos las fórmulas que se generan para los distintos tipos de consultas SQL que tratamos (ver sección de sintaxis permitida 2.3):

- **Consultas básicas:** En las consultas básicas las fórmulas están asociadas a la condición que aparece en la sección `WHERE`.

  Cuando una vista V esta definida mediante una consulta básica de la forma:

  `CREATE VIEW V AS`
  `SELECT` $e_1, \ldots, e_n$
  `FROM` $R_1, ..., R_m$
  `WHERE` C

entonces:

$\Phi(V) = \{|\varphi_1 \wedge ... \wedge \varphi_m \ \wedge \psi(C,\mu) \wedge s_Q(\mu)|(\varphi_1,\nu_1) \in \Phi(R_1),...,(\varphi_m,\nu_m) \in \Phi(R_m), \mu = \nu_1^{B_1} \odot ... \odot \nu_m^{B_m}|\}$

Donde la fórmula de primer orden $\psi(C,\mu)$ está definida de la siguiente forma:

- Si C $\equiv$ e, siendo e una expresión sin subconsultas, entonces $\psi(C,\mu) = C_\mu \wedge \bigwedge_{i=1}^{k}(A_i \neq null)$ con $A_i \in$ Att(C), donde Att(C) es el conjunto de atributos que aparecen en la condición C.

- Si C $\equiv C_1$ and C_2, entonces $\psi(C,\mu) = \psi(C_1,\mu) \wedge \psi(C_2,\mu)$

- Si C $\equiv C_1$ or C_2, entonces $\psi(C,\mu) = \psi(C_1,\mu) \vee \psi(C_2,\mu)$

- Si C $\equiv not(C_1)$, entonces $\psi(C,\mu) = \neg\psi(C_1,\mu)$

- `INNER JOIN`**:** Hacer un `INNER JOIN` de dos tablas sobre dos de sus campos es equivalente a hacer una consulta básica que contenga las dos tablas en el `FROM` y la condición del `ON` en el `WHERE`.

  Para consultas de la forma:

  `CREATE VIEW V AS`
  `SELECT` $e_1, \ldots, e_n$
  `FROM` R_1 `INNER JOIN` R_2 `ON` C_j `...  INNER JOIN` R_n `ON` C_n
  `WHERE` C_w
  `GROUP BY` $A_1, \ldots, A_n$
  `HAVING` C_h

  $\Phi(V) = \Phi(V')$, donde V' es de la forma:

  `CREATE VIEW V' AS`
  `SELECT` $e_1, \ldots, e_n$
  `FROM` $R_1, R_2, \ldots, R_n$
  `WHERE` C_w `and` C_j `and ...  and` C_n
  `GROUP BY` $A_1, \ldots, A_n$
  `HAVING` C_h

- **Operador** `UNION`: El resultado de utilizar el operador `UNION` entre dos consultas devuelve todas las tuplas obtenidas en la primera consulta junto con todas las tuplas obtenidas en la segunda. Con que en una de las consultas se obtengan tuplas es suficiente, dado que ya existiría un caso de prueba positivo para la unión de ambas.

Consideremos dos consultas cuales quiera C_1 y C_2. Para cada consulta tendremos un conjunto $\Phi(C_1) = \{(\varphi_{11}, \mu_1)...(\varphi_{1k_1}, \mu_n)\}$ y $\Phi(C_2) = \{(\psi_{21}, \mu_1)...(\psi_{2k_2}, \mu_n)\}$ de pares fórmula-tupla.

Si la vista V está definida de la siguiente forma:

```
CREATE VIEW V AS
C1
UNION
C2
```

Entonces: $\Phi(V) = \Phi(V1) \cup \Phi(V2)$, siendo $\cup$ la unión multiconjunto, `V1 = CREATE VIEW V1 AS C1` y `V2 = CREATE VIEW V2 AS C2`.

- **Operador** `INTERSECT`: El resultado de utilizar el operador `INTERSECT` entre dos consultas es la intersección entre el conjunto de tuplas devuelto por cada una de ellas.

  Si la vista V está definida de la siguiente forma:

  ```
  CREATE VIEW V AS
  C1
  INTERSECT
  C2
  ```

  Entonces: $\phi(V) = \{(\varphi, \mu) | \varphi = (\varphi_{11} \vee ... \vee \varphi_{1k_1}) \wedge (\psi_{21} \vee ... \vee \psi_{2k_2}), (\varphi_{11}, \mu_1) \in \Phi(V_1), ..., (\varphi_{1k_1}, \mu_n) \in \Phi(V_1), (\varphi_{21}, \mu_1) \in \Phi(V_2), ..., (\varphi_{2k_2}, \mu_n) \in \Phi(V_2)\}$, con `V1 = CREATE VIEW V1 AS C1` y `V2 = CREATE VIEW V2 AS C2`.

- `GROUP BY:` La cláusula `GROUP BY` en una consulta forma grupos entre los campos que aparezcan en ella. Una tupla pertenecerá al mismo grupo que otra si tienen el mismo valor en los campos que aparecen en él.

  Sea V la siguiente vista:

  ```
  CREATE VIEW V' AS
  SELECT e1,...,en
  FROM R1,...,Rn
  WHERE Cw
  GROUP BY e1,...,en
  ```

  Definimos:
  $P = \{|(\psi, \mu)|(\psi_1, \nu_1), ..., (\psi_m, \nu_m) \in (\theta(R_1 \text{ x } ... \text{ x } \theta(R_m)\psi = \psi_1 \wedge ... \wedge \psi_m, \mu = \nu_1^{B_1} \odot ... \odot \nu_m^{B_m}|\}$

  $\Phi(V) = \{| \bigwedge \Pi_1(A) \wedge \text{ aggreg(Q,A)}, s_Q(\Pi_2(A)) | A \subseteq P\}$
  Con $\Pi_1(A) = \{|\psi|(\psi, \mu) \in A|\}$ y $\Pi_2(A) = \{|\mu|(\psi, \mu) \in A|\}$

  aggregate(Q,A)=group(Q,Π_2(A)) $\wedge$ maximal(Q,A) $\wedge \varphi(C_h, \Pi_2(A))$
  group(Q,S) = $(\bigwedge |\varphi(C_w, \mu)|\mu \in S|) \wedge (\bigwedge \{|(e'_1)\nu_2 \wedge ... \wedge (e'_k)\nu_1 = (e'_k)\nu_2 | \nu_1, \nu_2 \in S|\})$

maximal(Q,A) $= \bigwedge\{|\neg\psi \vee \neg group(Q, \Pi_2(A)) \cup \{|\nu|\}|(\psi, \nu) \in (P - A)|\}$

Consideremos el siguiente ejemplo:

```
CREATE VIEW v AS
SELECT *
FROM t
WHERE a > 0
GROUP BY a
```

Donde T es una tabla con dos campos a, b y tres tuplas $\{[a_1, b_1], [a_2, b_2], [a_3, b_3]\}$. Al agrupar por a tendremos que hacer grupos con ese campo, todos los posibles grupos que se pueden dar para T se generan si: $a_1 = a_2 = a_3$ (dando lugar a un solo grupo de uno), $a_1 = a_2 \neq a_3$, $a_1 = a_3 \neq a_2$ y $a_2 = a_3 \neq a_1$ (dando lugar a tres grupos de dos) y $a_1 \neq a_2 \neq a_3$ (dando lugar a tres grupos de uno). Siguiendo ese razonamiento, si $\Phi(T) = \{(\phi_1, [a_1, b_1]), (\phi_2, [a_2, b_2]), (\phi_3, [a_3, b_3])\}$ la fórmula para v será:

$\Phi(v) = \{(\phi_1 \wedge (a_1 > 1) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_1 \neq a_2)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_1 \neq a_3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null))), [a_1, b_1]), (\phi_2 \wedge (a_2 > 1) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_2 \neq a_1)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_2 \neq a_3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null))), [a_2, b_2]), (\phi_3 \wedge (a_3 > 1) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_1 \neq a_2)) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_3 \neq a_1) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null))), [a_3, b_3]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_2 \wedge (a_2 > 1) \wedge (a_1 = a_2)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_3 \neq a_1) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null))), [a_1, b_1]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_1 = a_3)) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_2 \neq a_1) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null))), [a_1, b_1]), (\phi_2 \wedge (a_2 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_2 = a_3)) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_1 \neq a_2) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null))), [a_2, b_2]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_2 \wedge (a_2 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_1 = a_2) \wedge (a_1 = a_3)), [a_1, b_1]))\}$

- **Cláusula `HAVING`:** La cláusula `HAVING` afecta a los grupos generados por el `GROUP BY`. Nosotros tratamos las funciones de agregación `SUM()` y `COUNT()`. La función `SUM()` realiza el sumatorio de valores de un campo dentro de un mismo grupo y la función `COUNT()` cuenta el número de valores de un campo dentro de un mismo grupo.

Veamos dos ejemplos que extienden la vista v del ejemplo anterior con una cláusula `HAVING` y las funciones de agregación `SUM()` y `COUNT()`:

```
CREATE VIEW v1 AS
SELECT *
FROM t
WHERE a > 0
GROUP BY a
HAVING SUM(b)=3
```

```
CREATE VIEW v2 AS
SELECT *
FROM t
WHERE a > 0
GROUP BY a
```

`HAVING COUNT(b)=3`

La condición `SUM(b)=3` de la sección `HAVING` de la vista v1 nos fuerza a que en los grupos generados la suma de los valores del campo b de la tabla T sea igual a tres. Existen las siguientes posibilidades dependiendo de como se formen los grupos: si $a_1 = a_2 = a_3$, entonces $b_1 + b_2 + b_3 = 3$, si $a_1 = a_2 \neq a_3$, entonces $b_1 + b_2 = 3$ o $b_3 = 3$, si $a_1 = a_3 \neq a_2$, entonces $b_1 + b_3 = 3$ o $b_2 = 3$ y si $a_2 = a_3 \neq a_1$, entonces $b_2 + b_3 = 3$ o $b_1 = 3$, por último si $a_1 \neq a_2 \neq a_3$, entonces $b_1 = 3$ o $b_2 = 3$ o $b_3 = 3$.

Como en el ejemplo anterior si $\Phi(T) = \{(\phi_1, [a_1, b_1]), (\phi_2, [a_2, b_2]), (\phi_3, [a_3, b_3])\}$ la fórmula para $v1$ será:

$\Phi(v1) = \{(\phi_1 \wedge (a_1 > 1) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_1 \neq a_2)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_1 \neq a_3)) \wedge (b_1 = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_1 \neq null)), [a_1, b_1]), (\phi_2 \wedge (a_2 > 1) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_2 \neq a_1)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_2 \neq a_3)) \wedge (b_2 = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_2 \neq null)), [a_2, b_2]), (\phi_3 \wedge (a_3 > 1) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_1 \neq a_2)) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_3 \neq a_1)) \wedge (b_3 = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_3 \neq null)), [a_3, b_3]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_2 \wedge (a_2 > 1) \wedge (a_1 = a_2)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_3 \neq a_1)) \wedge (b_1 + b_2 = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_1 \neq null) \wedge (b_2 \neq null)), [a_1, b_1]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_1 = a_3)) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_2 \neq a_1)) \wedge (b_1 + b_3 = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_1 \neq null) \wedge (b_3 \neq null)), [a_1, b_1]), (\phi_2 \wedge (a_2 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_2 = a_3)) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_1 \neq a_2)) \wedge (b_2 + b_3 = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_3 \neq null) \wedge (b_2 \neq null)), [a_2, b_2]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_2 \wedge (a_2 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_1 = a_2) \wedge (a_1 = a_3) \wedge (b_1 + b_2 + b_3 = 3)) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_1 \neq null) \wedge (b_2 \neq null) \wedge (b_3 \neq null)), [a_1, b_1]))\}$

La condición `COUNT(b)=3` de la sección `HAVING` de la vista v2 nos obliga a que dentro de un grupo haya justo tres valores para el campo b, solo ocurre esto en el caso $a_1 = a_2 = a_3$ ya que dentro del grupo tenemos los tres valores b_1, b_2, b_3.

Para poder hacer nuestra fórmula para $\Phi(v2)$ necesitamos introducir $Card(C)$ como el cardinal de un conjunto C, en conclusion la fórmula de $v2$ será:

$\Phi(v2) = \{(\phi_1 \wedge (a_1 > 1) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_1 \neq a_2)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_1 \neq a_3)) \wedge (Card(\{b_1\}) = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_1 \neq null)), [a_1, b_1]), (\phi_2 \wedge (a_2 > 1) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_2 \neq a_1)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_2 \neq a_3)) \wedge (Card(\{b_2\}) = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_2 \neq null)), [a_2, b_2]), (\phi_3 \wedge (a_3 > 1) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_1 \neq a_2)) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_3 \neq a_1)) \wedge (Card(\{b_3\}) = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_3 \neq null)), [a_3, b_3]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_2 \wedge (a_2 > 1) \wedge (a_1 = a_2)) \wedge (\neg(\phi_3 \wedge (a_3 > 1)) \vee (a_1 \neq a_3)) \wedge (Card(\{b_1, b_2\}) = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_1 \neq null) \wedge (b_2 \neq null)), [a_1, b_1]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_1 = a_3)) \wedge (\neg(\phi_2 \wedge (a_2 > 1)) \vee (a_1 \neq a_2)) \wedge (Card(\{b_1, b_3\}) = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_1 \neq null) \wedge (b_3 \neq null)), [a_1, b_1]), (\phi_2 \wedge (a_2 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_2 = a_3)) \wedge (\neg(\phi_1 \wedge (a_1 > 1)) \vee (a_1 \neq a_2)) \wedge (Card(\{b_2, b_3\}) = 3) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_3 \neq null) \wedge (b_2 \neq null)), [a_2, b_2]), (\phi_1 \wedge (a_1 > 1) \wedge (\phi_2 \wedge (a_2 > 1) \wedge (\phi_3 \wedge (a_3 > 1) \wedge (a_1 = a_2) \wedge (a_1 = a_3) \wedge (Card(\{b_1, b_2, b_3\}) = 3)) \wedge ((a_1 \neq null) \wedge (a_2 \neq null) \wedge (a_3 \neq null) \wedge (b_1 \neq null) \wedge (b_2 \neq null) \wedge (b_3 \neq null)), [a_1, b_1]))\}$

Nótese que la única fórmula que se cumple es la última, que corresponde al grupo donde $a_1 = a_2 = a_3$.

- **Cláusula** `DISTINCT`**:** Cuando declaramos en una consulta sus elementos como `DISTINCT`, estamos forzando a que no nos devuelva valores repetidos. Esto mismo efecto se puede conseguir si agrupamos, usando `GROUP BY`, los elementos que aparecen en la sección `SELECT` de la consulta como `DISTINCT`, ya que al hacer grupos sobre sus valores sólo aparecerán una única vez.

  Generalizando, para consultas de la forma:

  `CREATE VIEW V AS`
  `SELECT DISTINCT` $e_1, \ldots, e_n$
  `FROM` $R_1, \ldots, R_n$
  `WHERE` C_w

  $\Phi(V) = \Phi(V')$, con V' de la forma:

  `CREATE VIEW V' AS`
  `SELECT` $e_1, \ldots, e_n$
  `FROM` $R_1, \ldots, R_n$
  `WHERE` C_w
  `GROUP BY` $e_1, \ldots, e_n$

Una vez generado el conjunto $\Phi(V) = \{(\varphi_i, \mu_i) | i = 1..n\}$ para la vista V, traducimos una a una cada una de las fórmulas φ_i del conjunto $\Phi(V)$, creando así n-problemas independientes de satisfactibilidad de restricciones.

Si el problema asociado a la fórmula φ_i es satisfactible, la instanciación de las variables constituirá un caso de prueba positivo para la vista V. En otro caso, si no es satisfactible, se prueba la satisfactibilidad del problema asociado a la fórmula φ_{i+1}.

Si ninguna de las fórmulas de $\Phi(V)$ es satisfactible, se generan de nuevo las fórmulas aumentando en uno el número de tuplas y se hace la misma comprobación. Así sucesivamente hasta llegar al número límite de tuplas permitido por el sistema.

Capítulo 3

Implementación

3.1 Investigación

Este proyecto se trata de una extensión del trabajo descrito en [4]. Por tanto, a causa de la naturaleza científica del mismo y antes de comenzar con la implementación del código de la herramienta, hizo falta realizar un proceso previo de investigación de algunos artículos existentes relacionados con el trabajo que íbamos a realizar. Para ello, también fue necesario refrescar conceptos de Bases de Datos, así como de la lógica de primer orden a la que íbamos a traducir las restricciones del lenguaje SQL, para generar las fórmulas objetivo de nuestro proyecto.

Conocíamos lo que son los casos de prueba, ya que los habíamos estudiado y utilizado durante la carrera para comprobar la corrección de las aplicaciones realizadas a lo largo de la misma. Sin embargo, estos casos de prueba habían sido siempre utilizados en aplicaciones software y no en consultas SQL. Por tanto, los casos de prueba que habíamos implementado eran algo distintos, ya que incorporaban un proceso manual y dependiente de cada aplicación en concreto, es decir, buscábamos "a mano" los puntos débiles donde las aplicaciones a probar podían fallar y ajustábamos las pruebas a dichos puntos débiles.

Por otra parte, al desconocer la existencia del sistema DES, fue necesario familiarizamos con él, con su implementación y observar cómo funcionaba. Comprobamos que este sistema permite hacer deducciones lógicas y responder a consultas, fundamentándose en los hechos y reglas almacenados en él.

Además, fue necesario estudiar el lenguaje de programación lógica utilizado para la implementación de la herramienta y sobre el que se basa el sistema DES: SICStus Prolog. Si bien conocíamos superficialmente el lenguaje Prolog al haber programado un poco con él en alguna que otra asignatura, supuso un cambio de mentalidad en nuestra manera de programar, ya que tuvimos que habituarnos a la programación recursiva, que es completamente diferente a la forma de programar iterativa a la que estábamos acostumbrados.

Asimismo, repasamos con este lenguaje algunas técnicas vistas en Metodología y Tecnología de la Programación, como la técnica de vuelta atrás (backtracking).

Por otro lado, también desconocíamos el funcionamiento de la programación lógica basada en restricciones, por lo que tuvimos que aprenderlo, para así poder traducir a lenguaje de restricciones

las fórmulas generadas con anterioridad en la lógica de primer orden. En el lenguaje de restricciones vimos conceptos nuevos para nosotros, como es el mecanismo de la *reificación* de restricciones el cual detallamos a continuación.

Una restricción reificada la escribimos de la siguiente forma:

Restricción #<=> B.

Donde B es una variable que contiene el resultado de valorar la expresión de la parte izquierda, que puede ser 0 si es falso o bien 1 si es cierto.

En el ejemplo: A #< 8 #<=> B. Si la variable A es menor que 8, B tomará el valor 1, en caso contrario tomará valor 0.

Así comprobamos que se pueden implementar conjunciones y disyunciones de restricciones de manera eficiente. Las conjunciones tienen esta forma: B1*...*Bn #<=> B, ya que en el momento en que alguna Bi, i=1,...,n, sea cero, toda la expresión se vuelve falsa. Las disyunciones tienen la forma siguiente: B1+...+Bn #<=> B para que en el momento en que alguna Bi, i=1,...,n, valga 1, toda la expresión sea cierta.

3.2 Planificación

Para realizar este proyecto hemos escogido el método de desarrollo incremental. Este tipo de desarrollo nos ha permitido, partiendo de una implementación sencilla que afrontara los requisitos básicos a tratar, ir evolucionando el sistema añadiendo nuevos requisitos, o mejorando lo que ya fueron completados anteriormente, en una serie de fases o *iteraciones* de forma que al final de cada fase tuviéramos una versión estable de la herramienta.

Esta metodología de desarrollo nos ha permitido realizar un seguimiento claro de los objetivos que se iban cumpliendo, así como sacar ventaja de lo que habíamos aprendido a lo largo del desarrollo anterior para las siguientes versiones del sistema.

A continuación se describen las fases en que se ha dividido el proceso de desarrollo.

1. *Fase de Iniciación:*

   En esta fase el objetivo principal era el desarrollo de un prototipo que mostrara el funcionamiento básico de la herramienta. Consideramos como funcionalidad básica la generación de fórmulas lógicas de primer orden que representan las restricciones de clave primaria y clave ajena de las tablas que forman la definición de la base de datos.

   Otro objetivo de la inicialización fue lograr imprimir las fórmulas generadas en un archivo que se utilizaría como método de depuración en iteraciones posteriores.

   Pero cabe destacar que la finalidad última de esta etapa fue familiarizarnos tanto con la herramienta DES, como con la forma de programar con Prolog.

2. *Segunda Fase:*

   El objetivo principal de esta fase fue obtener un caso de prueba positivo sobre vistas definidas con consultas simples. Para ello fue necesario añadir a las fórmulas de las restricciones generadas en la etapa anterior, aquellas correspondientes a las restricciones propias de la consulta SQL generadora de la vista como, por ejemplo, las condiciones establecidas en la cláusula `WHERE`. Una vez generadas las fórmulas el siguiente paso fue traducirlas a lenguaje de restricciones, es decir, reificarlas y enviarlas al resolutor que, al instanciar las variables involucradas, nos proporcionaría el caso de prueba buscado.

   Como en la fase anterior, otro objetivo fue crear un nuevo archivo, que también se utilizaría como método de depuración, en el que aparecieran las fórmulas generadas después de la instanciación de las variables.

3. *Tercera Fase:*

   A partir de esta fase, y en las sucesivas, el objetivo fue ir ampliando la sintaxis del lenguaje SQL admitida por la herramienta. Esta fase, en concreto, trató sobre la inclusión de los operadores SQL de conjuntos, `UNION` e `INTERSECT`, así como sobre la generación de las fórmulas correspondientes a este tipo de consultas, admitiendo el anidamiento sucesivo de las mismas.

4. *Cuarta Fase:*

   El objetivo de esta fase fue incluir en las consultas admitidas las sentencias `GROUP BY` y `HAVING`, así como sus operadores de agrupación `SUM` y `COUNT` y generar las fórmulas correspondientes.

   Fue en esta fase en la que encontramos el mayor de los inconvenientes en el proceso de implementación ya que, hasta este momento, el procedimiento que seguía la herramienta era el siguiente (ver Figura 3.1):

   (a) Para cada posible fila de cada una de las tablas involucradas en la vista μ_n, generar todas las fórmulas de restricciones φ_n que dicha fila debe satisfacer para formar parte del caso de prueba positivo, lo que conforma el conjunto Φ(v) para la vista.

   (b) Traducir a lenguaje de restricciones, usando reificación, cada una de las fórmulas generadas con lo que, para cada φ_n, obtenemos φ_n'.

   (c) Agrupar todas las fórmulas generadas como una disyunción de las mismas de manera que al evaluar dicha disyunción el resultado sea mayor o igual que uno, esto es, al menos una de las fórmulas se satisfaga. Usando reificación esto se consigue pidiendo que (φ_1'+...+φ_n') #<=> B con B #>= 1.

   (d) Enviar esta única fórmula, que agrupa a todas las demás, al resolutor de restricciones para obtener el caso de prueba buscado.

   El problema con el que nos encontramos al usar esta metodología fue que, con la sintaxis admitida hasta ese momento, las fórmulas generadas eran muy extensas, tanto que al enviar al resolutor la disyunción de todas ellas, éste era incapaz de abordar todo el trabajo que se

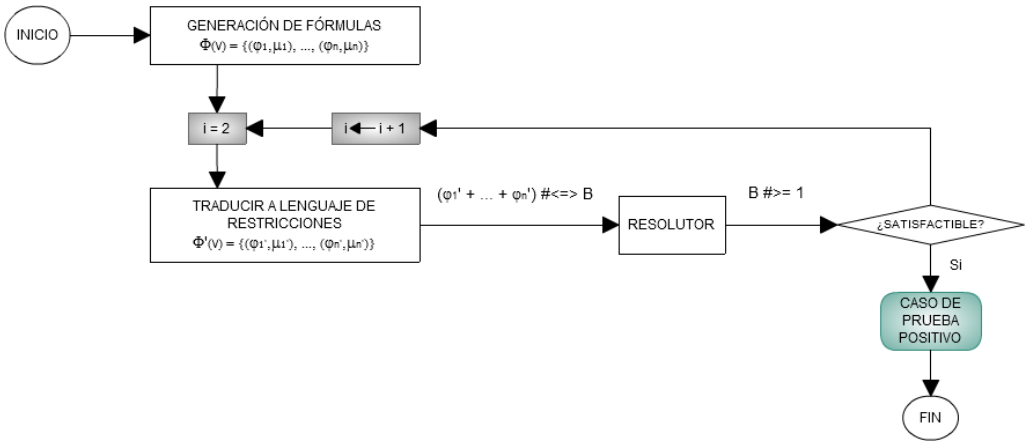

Figura 3.1: Esquema de la primera implementación seguida

le estaba requiriendo en un tiempo aceptable, dejando la herramienta bloqueada.

Ante este problema proponemos como solución cambiar el procedimiento de ejecución seguido de la siguiente manera (ver Figura 3.2):

(a) Una vez generado el conjunto ϕ(v) para la vista v, traducir a lenguaje de restricciones las fórmulas φ_j para cada posible fila de cada una de las tablas involucradas.

(b) Enviar una a una las fórmulas generadas al resolutor creando así n-problemas independientes de satisfactibilidad de restricciones.

(c) Si el problema asociado con la fórmula φ_j es satisfactible, la instanciación de las variables constituirá un caso de prueba para la vista v. En caso contrario, si es insatisfactible, se prueba la satisfactibilidad de la fórmula φ_{j+1}.

Siguiendo esta metodología no sólo conseguimos que la herramienta funcionara correctamente para los casos en que la consulta SQL tuviera `GROUP BY` y `HAVING`, sino que ganamos velocidad en general a la hora de analizar cualquier otra consulta.

5. *Quinta Fase:*

El objetivo principal de esta fase fue incluir el operador para calcular el producto cartesiano `INNER JOIN` así como la posibilidad de anidarlo, generando para ello las fórmulas oportunas.

Otro objetivo de esta fase fue incluir más información en el archivo usado para depuración dada la gran complejidad que tomaron las fórmulas generadas con los avances de la fase anterior. La información de más que creímos oportuno incluir es el resultado de la reificación (más coloquialmente hablando, el valor de su B asociada) para cada una de las fórmulas, para así poder comprobar cuál es la fórmula que se ha satisfecho para dar lugar al caso de prueba generado como salida del sistema. Al incluir toda esta información en el archivo, nos dimos cuenta de que no solamente podría ser usado como método de depuración, sino que podría ser considerado como una salida adicional del sistema para quien necesite más información

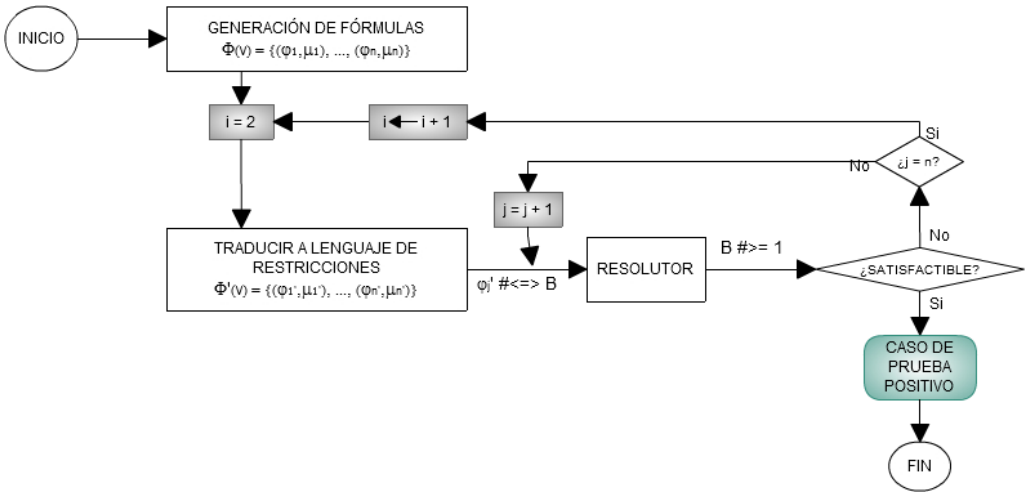

Figura 3.2: Esquema de la segunda implementación seguida

sobre la base de datos usada, aparte del caso de prueba en sí.

6. *Sexta fase:*

   El objetivo de esta fase fue incluir la cláusula `DISTINCT`, generando las fórmulas necesarias para eliminar las filas duplicadas en los resultados de una instrucción `SELECT`.

7. *Séptima fase:*

   La finalidad de esta última fase fue ampliar la funcionalidad del sistema para que permitiera el control de los valores nulos.

   Los nulos tienen una particularidad con respecto al resto de tipos: no pertenecen a ningún dominio de datos ya que no se consideran como un valor, sino que indican la ausencia del mismo. Como consecuencia, las comparaciones entre nulos no resultan ciertas o falsas, se puede considerar que su valor es desconocido. Por ejemplo, si a y b son nulos, expresiones como las siguientes: a = b, a <> b, a < b, a > b no tienen ningún sentido.

   Por todo esto, al usar un operador aritmético-lógico en las cláusulas `WHERE` o `HAVING`, añadimos a la fórmula generada la condición de que los campos que se están usando como operadores no puedan ser nulos. También se añade esta condición a los campos que así lo indican de forma explícita en su definición, es decir, los declarados como `NOT NULL` o `PRIMARY KEY`. Por otro lado, es posible que en una consulta se quieran obtener los registros que tengan campos cuyo valor sea igual a null, o distinto de null, por lo que en estos casos también añadimos las condiciones correspondientes.

   Una vez creadas las fórmulas, el siguiente paso fue traducirlas a lenguaje de restricciones. Nuestro propósito fue almacenar un estado para cada una de las variables, cuyo valor fuera 0

para indicar que no es nulo y 1 para indicar que sí lo es, de forma que en cualquier momento se pudiera consultar ese estado y/o modificarlo.

Para implementar esto, la idea inicial fue utilizar predicados dinámicos, ya que nos daban la posibilidad de añadir y eliminar variables en tiempo de ejecución, proporcionándonos de forma fácil y rápida cierta información sobre el estado de cada una de ellas. Sin embargo, este método no funcionó ya que las variables sin instanciar se sustituían por variables nuevas en el momento de la inserción, impidiendo la distinción de unas con otras.

Como alternativa, usamos una lista de tuplas donde guardamos todas las variables junto con su estado. A la hora de lanzar las restricciones de una variable, consultamos en la lista su variable de estado correspondiente y le aplicamos la restriccion de que sea nulo o no según corresponda y finalmente, hacemos el labeling tanto de las variables en sí mismas, como de las variables de estado.

3.3 Arquitectura

En esta sección describiremos como esta organizado el código encargado de la generación de los casos de prueba dentro de `DES`, empezando por una visión general para luego ir profundizando en las partes importantes.

Dando una visión general, el código se compone de siete predicados principales, los cuales enumeramos a continuación:

- `z_tc`
- `zeta_generator`
- `zetaStepTwoPK`
- `zetaStepTwoFK`
- `zetaStepThree`
- `zetaStepThreeNull`
- `traduccion`

Explicando la Figura 3.3: El encargado de generar e imprimir el caso de prueba es `z_tc`, que tratará de obtener una instancia válida de la vista empezando con una instancia con un número n de filas parametrizado por el usuario y aumentando progresivamente éste número en uno, si no ha encontrado solución, hasta llegar a un máximo fijado también por el usuario. Para generar esta instancia válida `z_tc` llama a `zeta_generator` con el número de tuplas que quiere insertar en las tablas y éste crea una instancia de la base de datos simbólica (es decir, un conjunto de tablas y vistas que intervienen en la prueba con sus campos sin instanciar) con ese número de tuplas y realiza el trabajo llamando a: `zetaStepTwoPK` (encargado de generar las fórmulas de clave primaria), `zetaStepTwoFK` (encargado de generar las fórmulas de clave ajena), `zetaStepThree` (encargado de generar las fórmulas de la vista), `zetaStepThreeNULL` (encargado de generar las fórmulas para el tratamiento de los nulos) y `traduccion` (encargado de traducir estas fórmulas a lenguaje de restricciones y ejecutar el resolutor para que busque valores válidos para instanciar las variables). Si `z_tc` llega hasta el número máximo de tuplas sin encontrar una solución le indicará al usuario

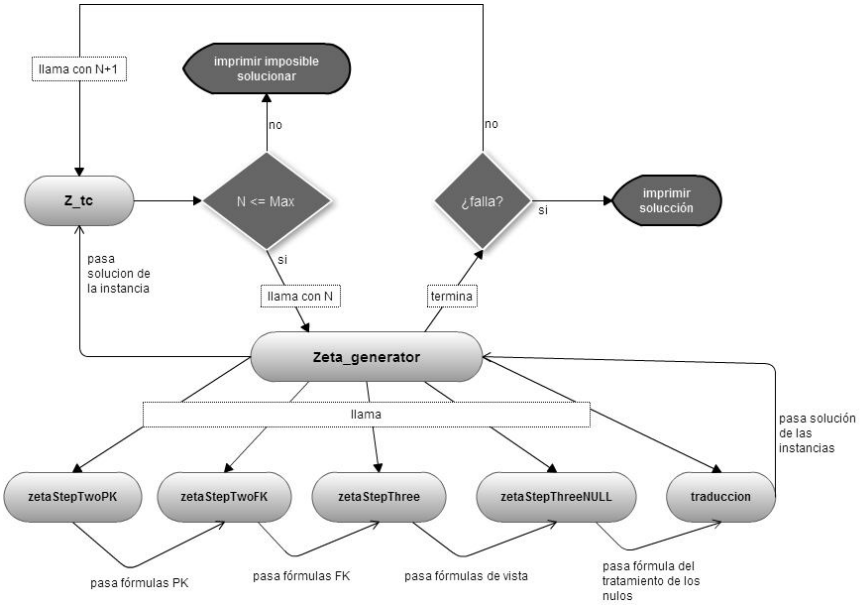

Figura 3.3: Visión general del funcionamiento del programa

que no se ha podido generar un caso de prueba para esa vista.

Aparte de la generación de fórmulas `zeta_generator` también se encarga de crear una lista con el estado de cada miembro de la instancia de la base de datos, para ello usa el predicado `creaEstados`. El estado de los miembros es una variable que nos indica si son o no nulos. Esta lista se utiliza en la traducción para lanzar las restricciones que incluyan el uso de nulos.

```
%crea la 'base de datos' de los estados
 creaEstados([],[]).
 creaEstados([X|Xs],[(X,V)|Zs]):-
  creaEstados(Xs,Zs).
```

creaEstados

Adicionalmente se han implementado predicados para poder acceder a esta lista de forma cómoda.

Veamos el código de `z_tc` y `zeta_generator`:

```
 z_tc(N, MainV, Dic, TestCase, LZetaTables, LZetaViews,Salida) :-
     tc_size(_,Max),
     M is Max + 1,
     M=N,
     write_log_list(['Info: No se puede generar el test case.',nl])
     ;
```

```
7       (
8           write('Prueba con test case de tamaño '),write(N),nl,
9           M is N+1,
10          (
11              (
12                  zeta_generator(N, MainV, Dic, TestCase, LZetaTables,
                        LZetaViews, Lt,Salida),
13                  !,
14                  write_log_list(['Info: Test case sobre enteros:',nl]),
15                  write_salida_null(Salida)
16              )
17              ;
18              z_tc(M, MainV,Dic, TestCase, LZetaTables, LZetaViews,_)
19          )
20      ).
```

z_tc

```
1   zeta_generator(InstanceSize, MainV, Dic, TestCase, LZetaFK, LZetaViews,
        Lt,Salida):-
2       tc_tables(T), %devuelve una lista T con el nombre de las tablas
3       tc_views(Views), %devuelve una lista Views con el nombre de las
            vistas
4       instancesOfTables(T,Lt, InstanceSize), %crea la base de datos
            simbólica
5       concat_lists(Lt,Lauxt),
6       concat_lists(Lauxt, ListT),
7       creaEstados(ListT,Lestados), %crea una lista con las variables y sus
             estados (0=no nulo, 1=nulo)
8       zetaStepTwoPK(T,Lt,LZetaTables,Lestados), %genera fomulas de claves
            primarias
9       zetaStepTwoFK(T, T, LZetaTables, LZetaFK), %genera fomulas de claves
             ajenas
10      deleteBesTables(LZetaFK,LZetaTotal),
11      zetaStepThree(LZetaTotal, Dic, Views, LZetaViews), %genera formulas
            para las vistas
12      zetaStepThreeNULL(LZetaViews,LZetaViewsNull), %genera las formulas
            para los campos not null
13  write_File(LZetaTables,LZetaViewsNull,0), %escribe en el fichero de
        debug
14  separaFormulasTabla(LZetaFK, TestCase), %separa las formulas de las
        filas para la impresion del TestCase
15  prepara_salida(TestCase,Lestados,Salida), %da como salida las variables
        de las tablas con sus estados
16  traduccion(LZetaViewsNull,Lt,Lestados, MainV,LZB,B), %lanza las
        restricciones de la vista principal
17      !,
18  ( B == 0 ->
19    (
20      write_File(LZetaFK,LZB,1), %fichero con los valores de las variables
             y de las B(depuracion).
21      !,
22      fail
23    )
```

```
24      ;
25      (
26          write_File(LZetaFK,LZB,1) %fichero con los valores de las variables
                y de las B(depuracion).
27      )
28  ).
```

zeta_generator

Implementación de las fórmulas

Prolog permite implementar las fórmulas que va a ir generando nuestro programa, de una forma bastante parecida a la natural y gracias a la recursividad se hace fácil su tratamiento posterior. Como se ha visto en la generación de fórmulas (ver Sección 2.4) estas se componen de los operadores lógicos: $\wedge, \vee, \neg$ y los operadores de comparación: $=, \neq, >, <, \leq, \geq$, adicionalmente hemos añadido dos operadores: is, is_not para indicar si una variable tiene que ser o no nula.

Para implementar los operadores lógicos hemos decido usar una notación prefija y sustituir los símbolos $\wedge, \vee, \neg$ por sus nombre en inglés quedando las estructuras: $and(\_,\_), or(\_,\_), not(\_)$. Para los operadores de comparación hemos implementado la estructura $bc(\_,\_,\_)$ *basic condition* o condición básica donde los tres campos son: una variable, un operador de comparación o un is/is_not para el caso de los nulos y una variable o "*null*".

A continuación veremos con más detalle el funcionamiento de los predicados `zetaStepTwoPK`, `zetaStepTwoFK` , `zetaStepThreeNULL` y `zetaStepThree` y cómo van generando las fórmulas para la vista en estudio.

- **zetaStepTwoPK:** Este predicado recibe como entrada una lista con todas las tablas a tratar y otra con la instancia simbólica generada por `zeta_generator` y devuelve una lista con cada una de las tuplas de la instancia acompañada de su fórmula de clave primaria y de los campos no nulos.

  Como podemos ver en la Figura 3.4 `zetaStepTwoPK` usa `comparacionNew` y éste a su vez `comparacionOr` para generar las fórmulas. El primero de ellos se encarga de coger los campos que sean clave primaria de la instancia simbólica para pasarlos a `comparacionOr` que es el encargado de ir añadiendo las restricciones para que no se repitan, teniendo en cuenta que si la clave es de más de un elemento solo tiene que ser uno de ellos diferente. Una vez obtenida la fórmula, el predicado `simplifica` se encarga de simplificar la fórmula quitando las partes redundantes, siguiendo estas normas: $and(X, true) = X$ y $or(X, false) = X$.

  Las fórmulas así generadas se pasan a `trataTuplaNull` que se encarga de añadir las fórmulas para las variables declaradas como *not null* o *primary key* en la definición de la tabla.

```
1           zetaStepTwoPK([],_,[],_).
2           zetaStepTwoPK([T|Rtables],[Lt|Rlt],[VueltaT|Rzpk],Lestados)
                :-
3             get_primary_key(T,Lpk),
4             get_not_nullables(T,NotNull),
5
6             attr_name_2_attr_pos_list(Lpk,T,Lpos),
7             selecPk(Lpos,Lt,ClavesPrim),
```

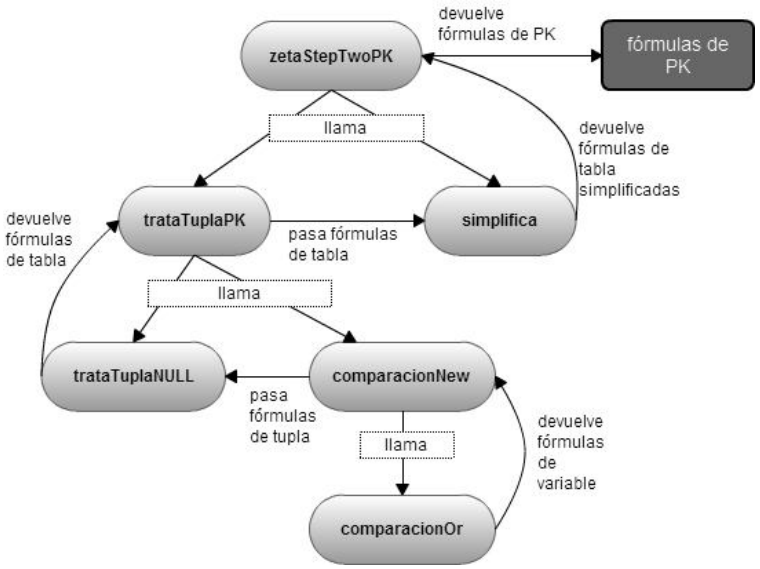

Figura 3.4: Llamadas realizadas por zetaStepTwoPK

```
8
9               attr_name_2_attr_pos_list(NotNull,T,LposNull),
10              selecPk(LposNull,Lt,ListNull),
11
12              trataTuplaPK(Lt,ListNull,ClavesPrim,ClavesPrim,Vuelta),
13              simplifica(Vuelta,VueltaSS),
14                 applyBes(VueltaSS, VueltaS,Lestados),
15              VueltaT=..[T,VueltaS],
16              zetaStepTwoPK(Rtables,Rlt,Rzpk,Lestados).
```

zetaStepTwoPK

```
1           comparacionNew([],_,true).
2
3           comparacionNew([ClaveP|RClaveP],ClaveP2,Evuelta):-
4            ClaveP == ClaveP2,
5            comparacionNew(RClaveP,ClaveP,Evuelta).
6
7           comparacionNew([ClaveP|RClaveP],ClaveP2,and(Evuelta,VueltaOr
                 )):-
8            comparacionOr(ClaveP2,ClaveP,VueltaOr),
9            comparacionNew(RClaveP,ClaveP2,Evuelta).
```

comparacionNew

```
1           trataTuplaPK([],_,_,[],[]).
2           trataTuplaPK([TuplasT|RTuplasT],[TNull|RTNull],ClavesPrim,[
                 ClavePrim|RClavesPrim],[(Evuelta2,TuplasT)|Rvuelta2]):-
```

```
3               comparacionNew(ClavesPrim, ClavePrim, EvueltaPK),
4               trataTuplaNull(TNull, EvueltaNull),
5               Evuelta2 = and(EvueltaPK, EvueltaNull),
6               trataTuplaPK(RTuplasT, RTNull, ClavesPrim, RClavesPrim,
                    Rvuelta2).
```

trataTuplaPK

```
1               comparacionOr([],[], false).
2               comparacionOr([Cp1|RCp1],[Cp2|RCp2], or(bc(Cp1,'/=',Cp2),
                    VueltaOr)):-
3                   comparacionOr(RCp1,RCp2, VueltaOr).
```

comparacionOR

```
1           trataTuplaNull([], true).
2           trataTuplaNull([TNull|RTNull], and(bc(TNull,'is_not',null),
                Vuelta2)):-
3             trataTuplaNull(RTNull, Vuelta2).
```

trataTuplaNull

- **zetaStepTwoFK:** Este predicado recibe la instancia simbólica con las fórmulas generadas por `zetaStepTwoPK` y devuelve una instancia simbólica con la *and* de las fórmulas de clave primaria y las fórmulas de clave ajena (ver Figura 3.5).`zetaStepTwoFK` utiliza para generar estas fórmulas los predicados `applyAllFK` y `newZetasTable`. `applyAllFK` es el encargado de generar las fórmulas de clave ajena para una tabla, para ello llama al predicado `applyOneFK` para que cree la fórmula usando `constraintsBes`, que a su vez usando otros predicados auxiliares genera las fórmulas de cada tupla.

  Una vez obtenidas las fórmulas, se le pasan al predicado `newZetasTable` que se encarga de comprobar si alguna de las claves ajenas de la tabla se enlaza con otra, si es así también se encarga de tratarlas. Generadas las fórmulas son devueltas por `zetaStepTwoFK`.

```
1           zetaStepTwoFK([], _Tablas, _InstanciaTablas, []).
2           zetaStepTwoFK([T|Rtables], LTablas, ZetasTables, [Li|Rzfk])
                :-
3               get_foreign_key(T, LfkN),
4               LfkN == [],
5               !,
6               obtainInstance(LTablas, T, ZetasTables, Li),
7               zetaStepTwoFK(Rtables, LTablas, ZetasTables, Rzfk).
8
9           zetaStepTwoFK([T|Rtables], LTablas, ZetasTables, [Vuelta|
                Rzfk]):-
10              get_foreign_key(T, LfkName),
11              obtainInstance(LTablas, T, ZetasTables, Li),
12              attr_name_2_attr_pos_list_fk(LfkName, T, Lfknumber),
13              applyAllFK(LTablas, Li, Lfknumber, ZetasTables, Vuelta),
14              newZetasTables(ZetasTables, Vuelta, NL),
15              zetaStepTwoFK(Rtables, LTablas, NL, Rzfk).
```

zetaStepTwoFK

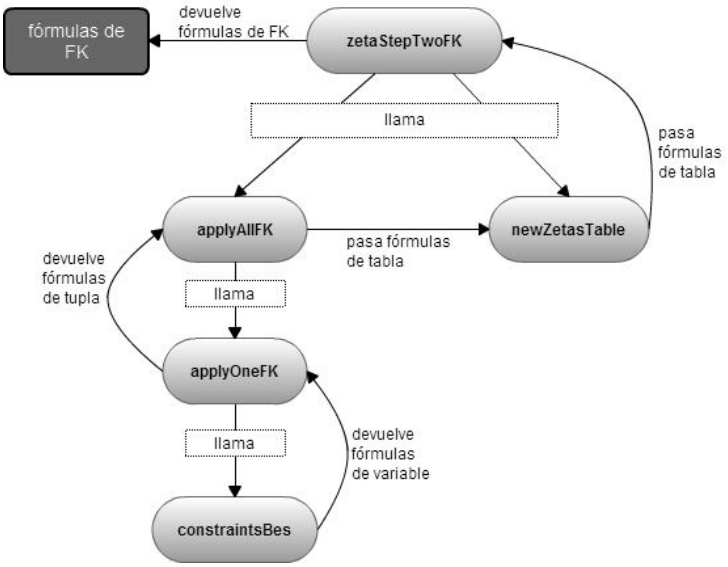

Figura 3.5: Llamadas realizadas por zetaStepTwoFK

```
1  applyAllFK(_LTablas,LTuplas,[],ZetasTables,LTuplas).
2  applyAllFK(LTablas, LTuplasInicial , [FK|RestoFK],
       ZetasTables, Vuelta):-
3      applyOneFK(LTablas, LTuplasInicial , FK, ZetasTables,
           Vuelta1),
4      applyAllFK(LTablas, Vuelta1, RestoFK, ZetasTables, Vuelta
           ).
```

applyAllFK

```
1  applyOneFK(LTablas, LTuplasInicial , (LAi, TReferences, LAf)
       , ZetasTables, Vuelta1):-
2      obtainInstance(LTablas, TReferences, ZetasTables,
           LTuplasfinal),
3      buidPartBes(LAi,LAf, LTuplasInicial, ClaveAj1,
           LTuplasfinal, ClavesAj2),
4      constraintsBes(ClaveAj1, ClavesAj2, Evuelta2),
5      buidallBes(LTuplasInicial, Evuelta2, Vuelta1).
```

applyOneFK

- **zetaStepThree:** Como se puede ver en la Figura 3.6 este predicado recibe la instancia generada por `zetaStepTwoFK` y devuelve una instancia de la base de datos junto con las fórmulas para las vistas a tratar. Para ello le pasa cada una de las vistas al predicado `zetaViews` que hace la fórmula para cada una de ellas, usando el predicado `trataConsulta`, que dependiendo de los campos de la consulta dentro de la vista, `WHERE, GROUP BY, DISTINCT`, etc (ver

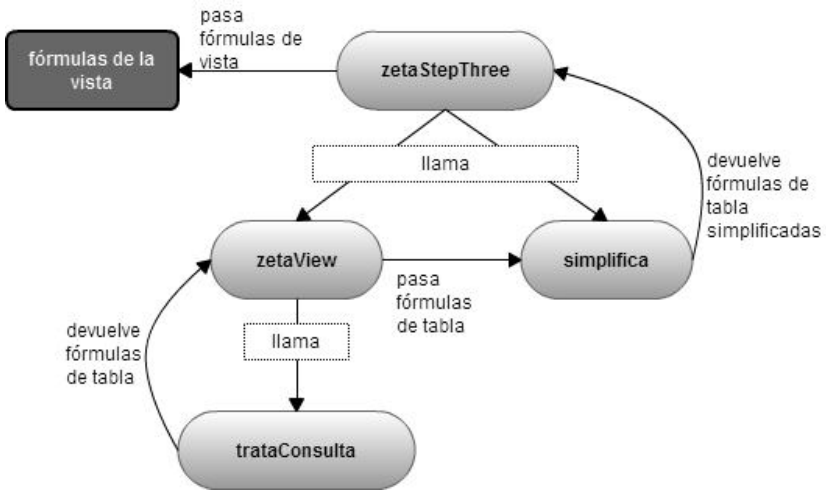

Figura 3.6: Llamadas realizadas por zetaStepThree

Sección: Sintaxis permitida 2.3), genera una fórmula para cada uno de ellos, las añade y se las devuelve a `zetaView`. Por último, las fórmulas se pasan al predicado simplifica que las devuelve simplificadas a `zetaStepThree`.

```
1          zetaStepThree(_ZetaTables, _Dic, [], []).
2          zetaStepThree(ZetaTables, Dic, [V|Rviews], [ZetaV|ZetaRviews
               ]):-
3              zetaView(ZetaTables, Dic, V, LZV),
4              simplifica(LZV, LZVSimp),
5              ZetaV=..[V,LZVSimp],
6              append(ZetaTables, [ZetaV], AllZetas),
7              zetaStepThree(AllZetas, Dic, Rviews, ZetaRviews).
```

zetaStepThree

```
1          zetaView(AllZ, _Dic, V, ZetaV):-
2              my_view('$des',V,_Arity, SQLst,_D,_L,_SCs,_,_),
3            trataConsulta(SQLst,AllZ,ZetaV).
```

zetaView

```
1          /** trataConsulta: Predicado auxiliar que recibe como
               entrada una
2          consulta y devuelve sus formulas asociadas en ZetaV **/
3
4          %Consulta con INNER JOIN con o sin group by
```

```
5           trataConsulta(Cons, AllZ, ZetaV):-
6
7               Cons = select(_,     % ALL or DISTINCT
8                     _,     %top(all)
9                           ExpList,    % es * o una lista de
                                expresiones
10                    from([(inner_join(Tabla1, Tabla2, Comparacion), _B)|
                          _RestoInnerJoin]),
11                          %En Tabla1 puede haber otro inner_join o
                                una lista con los campos de la
                               tabla
12                          where(CondicionWhere),
13                          group_by(G),
14                    having(Pred_H),
15                    order_by(_Lo1, _Lo2)
16                  ),
17              %Predicado auxiliar que devuelve todas las tablas en una
                    unica lista (Rels)
18                %y todos las comparaciones  que haya dentro de Tabla1 en
                     ComparacionesT1 (por si hay mas de un inner_join)
19                trata_inner_join(Tabla1, Tabla2, Rels, ComparacionesT1),
20                %Devuelve en Pred_W una lista con la comparacion del
                    Where, más la del primer
21                %inner_join y la devuelta en la llamada anterior
22              simpliFi(and(CondicionWhere, and(Comparacion,
                    ComparacionesT1)), Pred_W),
23
24              (G \= [] ->
25                 trataConsultaConGroupBy(AllZ, ExpList, Rels, Pred_W, G,
                        Pred_H, ZetaV)
26              ;
27                 trataConsultaSinGroupBy(AllZ, ExpList, Rels, Pred_W, ZetaV)
28              ).
29
30          %Una union en la consulta
31          trataConsulta(union(_, Cons1, Cons2), AllZ, ZetaV):-
32            trataConsulta(Cons1, AllZ, ZetaV1),
33            trataConsulta(Cons2, AllZ, ZetaV2),
34            append(ZetaV1, ZetaV2, ZetaV).
35
36          %Dos o mas uniones en la consulta
37          trataConsulta((union(_, Cons1, Cons2), _), AllZ, ZetaV):-
38            trataConsulta(Cons1, AllZ, ZetaV1),
39            trataConsulta(Cons2, AllZ, ZetaV2),
40            append(ZetaV1, ZetaV2, ZetaV).
41
42          %Una interseccion en la consulta
43          trataConsulta(intersect(_, Cons1, Cons2), AllZ, ZetaV):-
44            trataConsulta(Cons1, AllZ, ZetaV1),
45            trataConsulta(Cons2, AllZ, ZetaV2),
46            funcionIntersect(ZetaV1, ZetaV2, ZetaV).
47
48          %Dos o mas intersecciones en la consulta
49          trataConsulta((intersect(_, Cons1, Cons2), _), AllZ, ZetaV):-
```

```
50              trataConsulta(Cons1,AllZ,ZetaV1),
51              trataConsulta(Cons2,AllZ,ZetaV2),
52              funcionIntersect(ZetaV1,ZetaV2,ZetaV).
53
54            %Consulta con UNION o INTERSECT con o sin group by y
                  subconsulta con inner join
55            trataConsulta(Cons,AllZ,ZetaV):-
56
57              Cons = (select(_,       % ALL or DISTINCT
58                      _,      %top(all)
59                              ExpList,    % es * o una lista de
                                    expresiones
60                              from([(inner_join(Tabla1,Tabla2,
                                    Comparacion),_B)|_RestoInnerJoin]),
61                              where(CondicionWhere),
62                              group_by(G),
63                      having(Pred_H),
64                      order_by(_Lo1,_Lo2)
65                      ),
66                      _ % -> Esto es lo nuevo. Esta variable tiene esta
                            forma:['$t3'|_8780]
67                ),
68              %Predicado auxiliar que devuelve todas las tablas en una
                    unica lista (Rels) y todos las comparaciones
69                %que haya dentro de Tabla1 en ComparacionesT1 (por si
                    hay mas de un inner_join)
70              trata_inner_join(Tabla1,Tabla2,Rels,ComparacionesT1),
71                %Devuelve en Pred_W una lista con la comparacion del
                    Where, más la del primer inner_join
72                %y la devuelta en la llamada anterior
73              simpliFi(and(CondicionWhere,and(Comparacion,
                    ComparacionesT1)),Pred_W),
74
75              (G \= [] ->
76                  trataConsultaConGroupBy(AllZ,ExpList,Rels,Pred_W,G,
                        Pred_H,ZetaV)
77              ;
78                  trataConsultaSinGroupBy(AllZ,ExpList,Rels,Pred_W,ZetaV)
79              ).
80
81            %Consulta con UNION o INTERSECT con o sin group by y
                  subconsulta sin inner join
82            trataConsulta(Cons,AllZ,ZetaV):-
83
84              Cons = (select(_,       % ALL or DISTINCT
85                      _,      %top(all)
86                              ExpList,    % es * o una lista de
                                    expresiones
87                              from(Rels),
88                              where(Pred_W),
89                              group_by(G),
90                      having(Pred_H),
91                      order_by(_Lo1,_Lo2)
92                      ),
```

```
93                      _ % -> Esto es lo nuevo. Esta variable tiene esta
                            forma:['$t3'|_8780]
94                  ),
95
96              (G \= [] ->
97                  trataConsultaConGroupBy(AllZ,ExpList,Rels,Pred_W,G,
                        Pred_H,ZetaV)
98              ;
99                  trataConsultaSinGroupBy(AllZ,ExpList,Rels,Pred_W,ZetaV)
100             ).
101
102         %Consulta con o sin group by
103         trataConsulta(Cons,AllZ,ZetaV):-
104
105             Cons = select(Dist,     % ALL or DISTINCT
106                     _,     %top(all) ?¿?
107                             ExpList,    % es * o una lista de
                                    expresiones
108                             from(Rels),
109                             where(Pred_W),
110                             group_by(G),
111                     having(Pred_H),
112                     order_by(_Lo1,_Lo2)
113                      ),
114               % G=[] Pred_h = true if there is no groups
115             (Dist = distinct ->
116                 trataDistinct(ExpList,GDist),
117                 append(G,GDist,GNueva)
118             ;
119                 GNueva = G
120             ),
121             (GNueva \= [] ->
122                 trataConsultaConGroupBy(AllZ,ExpList,Rels,Pred_W,GNueva
                        ,Pred_H,ZetaV)
123             ;
124                 trataConsultaSinGroupBy(AllZ,ExpList,Rels,Pred_W,ZetaV)
125             ).
126
127         trataConsulta(_Cons,_AllZ,_ZetaV):-
128          raise_exception('Error al tratar la consulta: Formato de
                consulta incompatible').
```

trataConsulta

- **zetaStepThreeNull:** Este predicado añade las fórmulas para indicar que tuplas no pueden ser nulas. Para ello `zetaStepThreeNull` le pasa a `variablesFormulaNull` la tabla o vistas a tratar. Este utiliza los predicados `termVariables` ,que devuelve las variables que no pueden ser *null*, y `hacerFormulaNulos` que crea la fórmula $bc(var,'is\_not',null)$ para cada una de ellas. Una vez obtenida la fórmula se pasa a `simplifica` que devuelve la fórmula simplificada (ver Figura 3.7).

```
1           zetaStepThreeNULL([],[]).
2           zetaStepThreeNULL([Z|RZetas],[ZNull|RZetasNull]):-
```

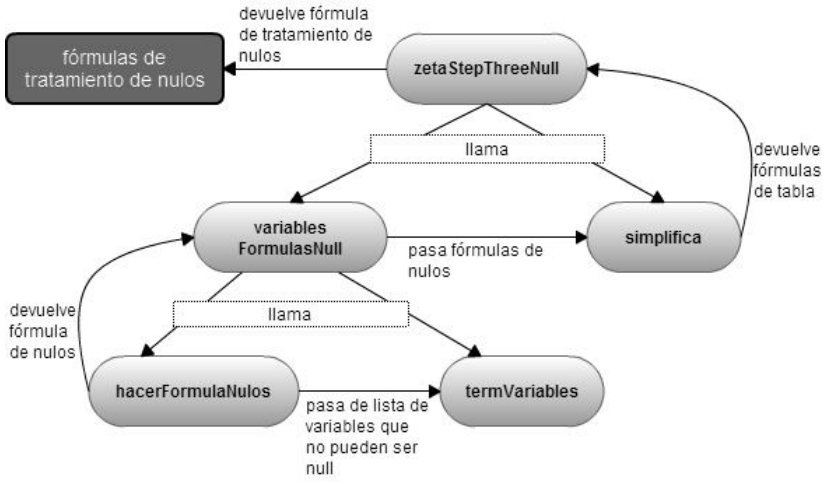

Figura 3.7: Llamadas realizadas por zetaStepThreeNull

```
3               Z=..[T,Tuplas],
4               variablesFormulasNull(Tuplas,TuplasNull),
5               simplifica(TuplasNull,TuplasNullS),
6               ZNull=..[T,TuplasNullS],
7               zetaStepThreeNULL(RZetas,RZetasNull).
8       \begin{lstlisting}
9
10      \begin{lstlisting}[style=Prolog,title={variablesformulaNull}]
11        variablesFormulasNull([],[]).
12        variablesFormulasNull([(Form,Tupla)|RVars],[(and(Form,
              FormNull1),Tupla)|Rzetas]):-
13              termvariablesForm(Form,[],LVars),
14              hacerFormulaNulos(LVars,FormNull1),
15              variablesFormulasNull(RVars,Rzetas).
```

zetaStepThreeNull

```
1         hacerFormulaNulos([],true).
2         hacerFormulaNulos([X|Rx],and(bc(X,'is_not',null),Rv)):-
3           hacerFormulaNulos(Rx,Rv).
```

hacerFormulaNulos

```
1         termvariablesForm(and(C1,C2),LVantes,LVdespues):-
2               termvariablesForm(C1,LVantes,LVd1),!,
3               termvariablesForm(C2,LVd1,LVdespues),!.
4
```

```
5           termvariablesForm(or(C1,C2),LVantes,LVdespues):-
6               termvariablesForm(C1,LVantes,LVd1),!,
7               termvariablesForm(C2,LVd1,LVdespues),!.
8
9           termvariablesForm(not(C1),LVantes,LVdespues):-
10              termvariablesForm(C1,LVantes,LVdespues),!.
11
12          termvariablesForm(bc(_C1,'=',C2),LVantes,LVantes):-C2==null
                ,!.
13          termvariablesForm(bc(_C1,'is',C2),LVantes,LVantes):-C2==null
                ,!.
14          termvariablesForm(bc(_C1,'is_not',C2),LVantes,LVantes):-C2==
                null,!.
15          termvariablesForm(bc(_C1,'/=',C2),LVantes,LVantes):-C2==null
                ,!.
16          termvariablesForm(bc(_C1,'<>',C2),LVantes,LVantes):-C2==null
                ,!.
17          termvariablesForm(bc(C1,_Op,C2),LVantes,LVdespues):-
18            var(C1),
19            var(C2),
20            addVar(LVantes,C1,LV1),!,
21            addVar(LV1,C2,LVdespues),!.
22          termvariablesForm(bc(_C1,_Op,C2),LVantes,LVdespues):-
23            var(C2),
24            addVar(LVantes,C2,LVdespues),!.
25          termvariablesForm(bc(C1,_Op,_C2),LVantes,LVdespues):-
26            var(C1),
27            addVar(LVantes,C1,LVdespues),!.
28
29          termvariablesForm(_form,LVantes,LVantes).
```

termvariablesForm

- **Traduccion:** Este predicado se encarga de traducir nuestras fórmulas a restricciones y de llamar al resolutor para que obtenga una instancia válida de la base de datos. Como se puede ver en la Figura 3.8, el predicado `traduccion` se encarga de buscar la vista para la cual se va a generar un caso de prueba entre todas las que hay. Una vez encontrada se pasa a `traduceVista` que pone el dominio a las variables que va a usar el *labeling*, llama a `tradRestriccion`, que es el encargado de traducir, y por último hace dicho *labeling* usando el predicado `my_fd_labeling`. Cuando termina su ejecución, devuelve en una variable (B) si se han cumplido o no las restricciones.

```
1           traduccion([],_,_,_,_,0).
2           traduccion([LZetaTables|RLZetaTables],Lt,Lestados,MainV,[LZB
                ],B):-
3             LZetaTables =..[T,LZ],
4             ( T == MainV ->( concat_lists(Lt,Lauxt),
5               concat_lists(Lauxt, ListT),
6               tc_domain(Min, Max), %recupera el dominio
7               my_fd_domain(ListT, Min, Max), %aplica el dominio
8
9               traduceVista(LZ,LB,ListT,Lestados,B),
10            )
```

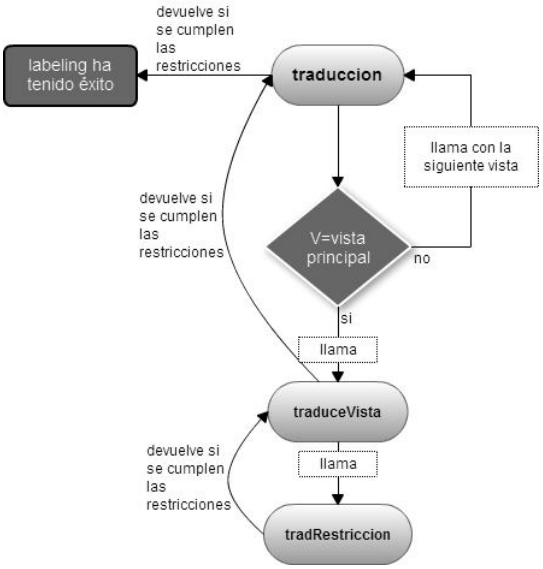

Figura 3.8: Llamadas realizadas por traduccion

```
11              ;
12              (traduccion(RLZetaTables,Lt,Lestados,MainV,[LZB],B))
13                ).
```

traduccion

```
1           traduceVista([],[],_,_,0).
2           traduceVista([(Formula,Fila)|_RForFila],[(Formula,Fila,B1,
                ParV)|_RForFila2],ListT,Lestados,1):-
3             dameEstadoFila(Fila,Lestados,ParV),
4             tradRestriccion(Formula,B1,Lestados),
5
6             (var(B1) ->
7               (B1 #>=1)
8             ;
9               %Si B1 vale cero, ya no hace falta hacer labeling, se
                    sabe que no se cumple
10              (B1 = 0 -> fail)
11            ),
12
13            dameVariables(Lestados,ListT,Estados),%obtiene los estados
                  para hacerles el labeling
14            my_fd_domain(Estados,0,1), %les aplica su dominio
15            my_fd_labeling(ListT),%hace el labeling de las variables
                  de la instancia simbolica
16            my_fd_labeling(Estados).
17
```

```
18          traduceVista([(Formula,Fila)|RForFila],[(Formula,Fila,0)|
                RForFila2],ListT,Lestados,B):-
19            traduceVista(RForFila,RForFila2,ListT,Lestados,B).
```

traduceVista

```
1           tradRestriccion(true,B,_):- B #= 1 .
2           tradRestriccion(false,B,_):- B #= 0 .
3           %NULOS: Estado = 0 -> NOT NULL // Estado = 1 -> NULL
4           tradRestriccion(bc(E1, 'is', null),B,Lestados):- dameEstado(
                E1,Lestados,V),(V #= 1)#<=> B.
5           tradRestriccion(bc(E1, 'is_not', null),B,Lestados):-
                dameEstado(E1,Lestados,V),(V #= 0)#<=> B.
6
7           tradRestriccion(bc(E1, '=', E2),B,Lestados):-
8                           ( E2==null ->(
9                               dameEstado(E1,Lestados,V),
10                              (V #= 1)#<=> B
11                          )
12                          ;
13                          (
14                              (E1 #= E2)#<=> B
15                          )
16                      ).
17
18          tradRestriccion(bc(E1, '<>', E2),B,Lestados):-
19                          tradRestriccion(bc(E1, '/=', E2),B,Lestados)
                                .
20
21          tradRestriccion(bc(E1, '/=', E2),B,Lestados):-
22                          ( E2==null ->(
23                              dameEstado(E1,Lestados,V),
24                              (V #= 0)#<=> B
25                          )
26                          ;
27                          (
28                              (E1 #\= E2) #<=> B
29                          )
30                      ).
31
32          tradRestriccion(bc(E1, '>=', E2),B,_Lestados):- (E1 #>= E2)
                #<=> B.
33          tradRestriccion(bc(E1, '=<', E2),B,_Lestados):- (E1 #=< E2)
                #<=> B.
34          tradRestriccion(bc(E1, '<=', E2),B,_Lestados):- (E1 #=< E2)
                #<=> B.
35          tradRestriccion(bc(E1, '>', E2),B,_Lestados):- (E1 #> E2)
                #<=> B.
36          tradRestriccion(bc(E1, '<', E2),B,_Lestados):- (E1 #< E2)
                #<=> B.
37
38          tradRestriccion(or(C1,C2),B,Lestados) :-
39                          tradRestriccion(C1, (B1),Lestados),
40                          tradRestriccion(C2, (B2),Lestados),
```

```
41                                  B #= B1+B2.
42          tradRestriccion(and(C1, C2),B,Lestados) :-
43                              tradRestriccion(C1, (B1),Lestados),
44                              tradRestriccion(C2, (B2),Lestados),
45                              B #= B1*B2.
46          tradRestriccion(not(and(C1,C2)),B,Lestados):-
47                  tradRestriccion(or(not(C1),not(C2)),B,Lestados).
48          tradRestriccion(not(or(C1,C2)),B,Lestados):-
49                  tradRestriccion(and(not(C1),not(C2)),B,Lestados).
50          tradRestriccion(not(bc(C1,OP,C2)),B,Lestados):-
51                  negacion(OP,NOP),
52                  tradRestriccion(bc(C1,NOP,C2),B,Lestados).
53          tradRestriccion(not(true),B,Lestados):-
54                  tradRestriccion(false,B,Lestados).
55          tradRestriccion(not(false),B,Lestados):-
56                  tradRestriccion(true,B,Lestados).
```

traduceVista

Capítulo 4

Ejemplos de ejecución

En este capítulo presentamos algunos ejemplos de ejecución con el fin de mostrar la mayor parte de las características implementadas en la herramienta, y de esta forma ayudar a comprender su funcionamiento, previamente explicado.

Sea un esquema de la base de datos con las siguientes tablas:

```
CREATE OR REPLACE TABLE r(a int, b int, c int NOT NULL, PRIMARY KEY(a,b));
CREATE OR REPLACE TABLE s(a int PRIMARY KEY, b int);
CREATE OR REPLACE TABLE t(a int PRIMARY KEY, b int, foreign key (a,b) references r(a,b));
```

Supongamos que se quiere generar un caso de prueba para diversas vistas con los siguientes parámetros:

- *Tamaño mínimo y máximo de la instancia:* Se refiere al número de tuplas que contiene cada una de las tablas. Por defecto el tamaño mínimo del caso de prueba 2 y máximo 7. El tamaño se puede modificar según se desee mediante el comando */tc_size N M*, donde N es el tamaño mímino y M el tamaño máximo
- *Dominio del caso de prueba:* Indica el rango de valores que pueden tomar cada uno de los campos de todas las tablas y vistas pertenecientes a la base de datos. Por defecto el dominio está comprendido entre 0 y 10. Éste se puede modificar mediante el comando */tc_domain N M*, donde N es el valor mínimo y M el valor máximo.

A continuación, mostramos algunos ejemplos.

4.1 EJEMPLO 1: Vista inicial sobre tabla

Sea la vista v definida como sigue:

```
CREATE OR REPLACE VIEW v(a,b) AS
SELECT s.a, s.b
FROM s
WHERE s.b = 8;
```

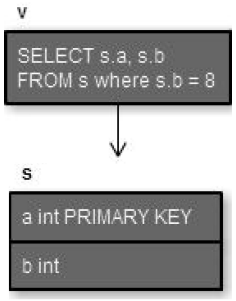

Figura 4.1: Esquema en árbol de vista de v

La vista v esta construida sobre una consulta sencilla que solo involucra una tabla del esquema de la base de datos. En la Figura 4.1 se puede ver su árbol de dependencias sintácticas.

Resultado obtenido:

s	
0	8
1	0

v	
0	8

Salida de DES:

Para generar un caso de prueba para v, basta con ejecutar la instrucción */test_case v* en DES. A continuación se muestra el resultado producido por el sistema. Como se puede observar, el caso de prueba generado es una instancia válida de la base de datos, donde la tabla s tiene dos tuplas.

```
DES-SQL> /test_case v
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|8|
|1|0|
```

Es posible introducir la instancia generada en las tablas involucradas de la base de datos mediante la siguiente instrucción: */test_case v add*.

Tras ejecutar la consulta v sobre la instancia generada, el resultado obtenido contiene una tupla, por lo que se trata de un caso de prueba válido.

```
DES-SQL> /test_case v add
DES-SQL> select * from v
answer(v.a:number(integer),v.b:number(integer)) ->
{
answer(0,8)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

$(((0 \neq 1) \wedge (8 = 8)) \wedge ((8 \neq null) \wedge ((1 \neq null) \wedge (0 \neq null))))$

4.2 EJEMPLO 2: Vista inicial sobre tabla y vista

Sea la vista w definida como sigue:

```
CREATE OR REPLACE VIEW w(a,b,c) AS
SELECT r.a, v.a, v.b
FROM r, v
WHERE r.b = v.b;
```

La vista w esta constituida no solo sobre una tabla, sino también sobre otra vista que, a su vez, depende de una tabla, haciendo el producto cartesiano de ambas. En la Figura 4.2 se puede ver el árbol de dependencias sintácticas de la vista w.

Resultado obtenido:

r		
0	8	0
0	0	0

t	
0	8
1	0

w		
0	0	8

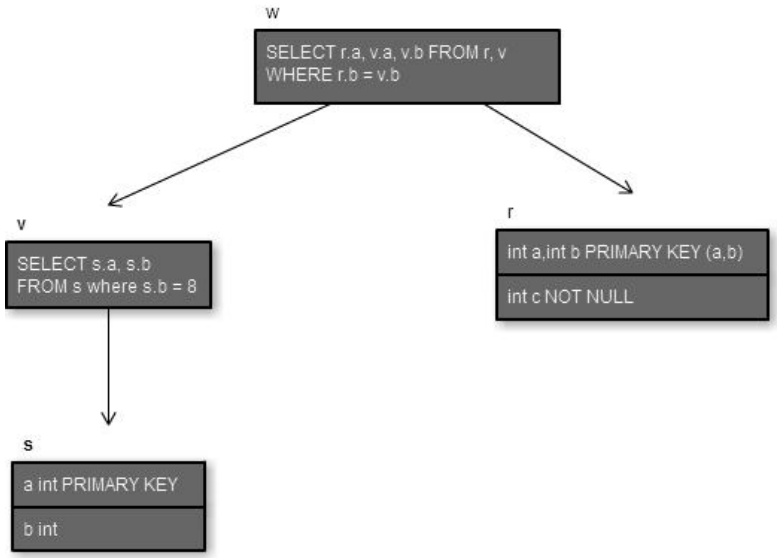

Figura 4.2: Esquema en árbol de vista de w

Salida de DES:

Como en este caso son dos las relaciones involucradas, el sistema deberá generar valores para ambas, siempre atendiendo a las restricciones del esquema de la base de datos y a las dependencias entre ellas establecidas en la consulta.

```
DES-SQL> /test_case w add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|8|
|1|0|
----------
r
-----------
|0|8|0|
|0|0|0|
```

Tras ejecutar la consulta w con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from w

answer(w.a:number(integer),w.b:number(integer),w.c:number(integer)) ->
{
answer(0,0,8)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

$((((((0 \neq 0) \vee (8 \neq 0)) \wedge (0 \neq null)) \wedge ((0 \neq 1) \wedge (8 = 8))) \wedge (8 = 8)) \wedge ((8 \neq null) \wedge ((1 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((0 \neq null) \wedge (0 \neq null)))))))$

El sistema intenta primero obtener un caso de prueba con el mínimo número de registros posibles, y sólo en el caso de que no lo pueda generar con este número, prueba a ver si lo consigue añadiendo un registro más. Así sucesivamente hasta llegar al límite máximo permitido. Es por ello por lo que el sistema muestra en pantalla lo siguiente "Prueba con test case de tamaño X". Por ejemplo, para v y w, consiguió obtener el caso de prueba buscado con tan sólo dos registros en cada tabla de la base de datos.

4.3 EJEMPLO 3: Vista sobre tabla con clave primaria y ajena

Sea la vista v1 definida como sigue:

```
CREATE OR REPLACE VIEW v1(a1,a2) AS
SELECT *
FROM t;
```

La vista v1 esta construida sobre una consulta sencilla que involucra solamente una tabla, en este caso t. Cabe destacar que la tabla a la que hace referencia contiene en su definición un atributo como clave primaria y que sus dos atributos están especificados como una clave ajena que apunta a la tabla r. En la Figura 4.3 se puede ver el árbol de dependencias sintácticas de la vista v1.

Resultado obtenido:

r		
0	0	0
1	0	0

t	
0	0
1	0

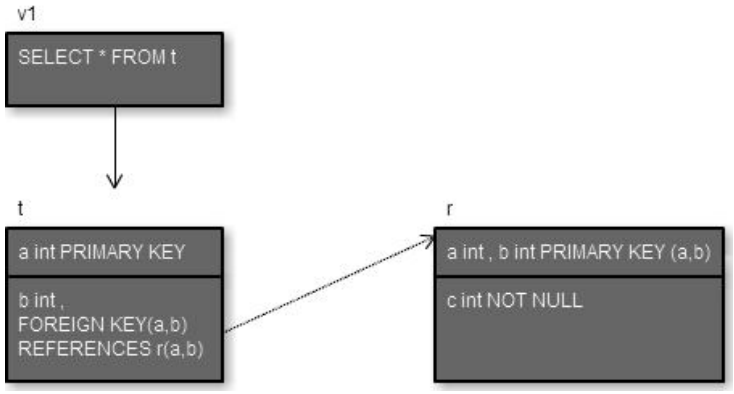

Figura 4.3: Esquema en árbol de vista de v1

v1	
0	0
1	0

Salida de DES:

En este caso, la salida del sistema mostrará datos para las dos tablas involucradas, aquella a la que la consulta hace referencia directamente, es decir t, y la que ésta referencia por medio de las claves ajenas que contiene en su definición, véase r.

```
DES-SQL> /test_case v1 add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
r
-----------
|0|0|0|
|1|0|0|
-----------
t
-----------
|0|0|
|1|0|
```

Tras ejecutar la consulta v1 con la instancia generada, el resultado obtenido contiene dos tuplas, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v1
answer(v1.a1:number(integer),v1.a2:number(integer)) ->
{
answer(0,0),
answer(1,0)
}
Info:  2 tuples computed.
```

Fórmulas obtenidas:

$(((0 \neq 1) \wedge (((((0 \neq 1) \vee (0 \neq 0)) \wedge (0 \neq null)) \wedge ((0 = 0) \wedge (0 = 0))) \vee ((((1 \neq 0) \vee (0 \neq 0)) \wedge (0 \neq null)) \wedge ((0 = 1) \wedge (0 = 0))))) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((1 \neq null) \wedge ((0 \neq null) \wedge ((1 \neq null) \wedge (0 \neq null)))))))$

4.4 EJEMPLO 4: Vista sobre una tabla y una vista con cláusula WHERE

Sea la vista v2 definida como sigue:

```
CREATE OR REPLACE VIEW v2(a1,a2,a3) AS
SELECT r.a, v.a, v.b
FROM r, v
WHERE r.b = v.b AND r.a > 2;
```

La vista v2 esta construida sobre una consulta sencilla que involucra a la tabla r y a la vista v, con una cláusula `WHERE` donde se pide que las tuplas de v y r cumplan ciertas restricciones. En la Figura 4.4 se puede ver el árbol de dependencias sintácticas de la vista v2.

Resultado obtenido:

r		
3	8	0
0	0	0

s	
0	8
1	0

v2		
3	0	8

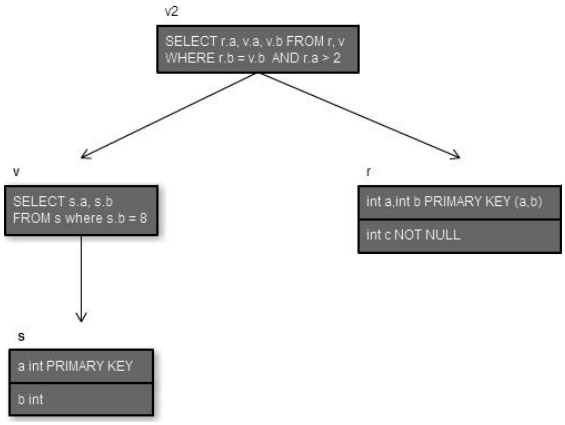

Figura 4.4: Esquema en árbol de vista de v2

Salida DES:

```
DES-SQL> /test_case v2 add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
r
-----------
|3|8|0|
|0|0|0|
-----------
s
-----------
|0|8|
|1|0|
```

Tras ejecutar la consulta v2 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v2
answer(v2.a1:number(integer),v2.a2:number(integer),v2.a3:number(integer)) ->
{
answer(3,0,8)
}
Info:  1 tuple computed.
```

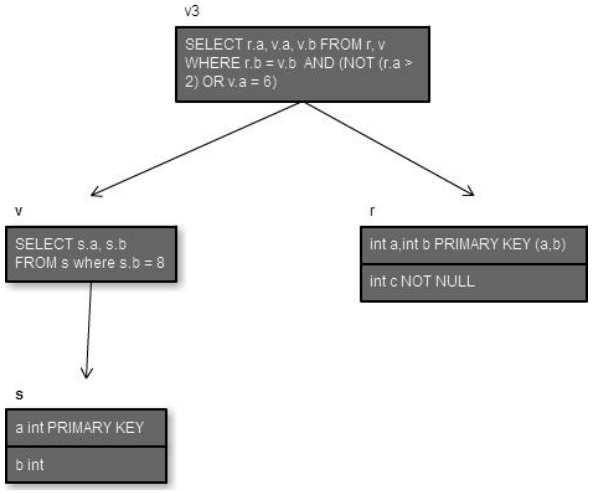

Figura 4.5: Esquema en árbol de vista de v3

Fórmulas obtenidas:

$((((((3 \neq 0) \vee (8 \neq 0)) \wedge (0 \neq null)) \wedge ((0 \neq 1) \wedge (8 = 8))) \wedge ((8 = 8) \wedge (3 > 2))) \wedge ((8 \neq null) \wedge ((1 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((0 \neq null) \wedge (3 \neq null)))))))$

4.5 EJEMPLO 5: Vista sobre una tabla y una vista con cláusula WHERE y todos los operadores lógicos

Sea la vista v3 definida como sigue:

```
CREATE OR REPLACE VIEW v3(a1,a2,a3) AS
SELECT r.a, v.a, v.b
FROM r, v
WHERE r.b = v.b AND (NOT (r.a > 2) OR v.a = 6);
```

La vista v3 esta construida sobre una consulta que involucra la tabla r y la vista v con una cláusula `WHERE` donde se pide que las tuplas de v y r cumplan ciertas restricciones usando todos los operadores lógicos permitidos. En la Figura 4.5 se puede ver el árbol de dependencias sintácticas de la vista v3.

Resultado obtenido:

r		
0	8	0
0	0	0

s	
0	8
1	0

v3		
0	0	8

Salida DES:

```
DES-SQL> /test_case v3 add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|8|
|1|0|
-----------
r
-----------
|0|8|0|
|0|0|0|
```

Tras ejecutar la consulta v3 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v3

answer(v3.a1:number(integer),v3.a2:number(integer),v3.a3:number(integer))
->
{
answer(0,0,8)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

$((((((0 \neq 0) \vee (8 \neq 0)) \wedge (0 \neq null)) \wedge ((0 \neq 1) \wedge (8 = 8))) \wedge ((8 = 8) \wedge ((0 \leq 2) \vee (0 = 6)))) \wedge ((8 \neq null) \wedge ((1 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((0 \neq null) \wedge (0 \neq null)))))))$

4.6 EJEMPLO 6: Vista con cláusulas GROUP BY y HAVING y función de agregación SUM

Sea la vista v4 definida como sigue:

```
CREATE OR REPLACE VIEW v4(a1) AS
SELECT r.b
FROM r, v
WHERE r.b = v.b
GROUP BY r.b
HAVING SUM(r.a) >= 4;
```

La vista v4 esta construida sobre una consulta que involucra a la tabla r y a la vista v con cláusulas:

- `WHERE` donde se pide que las tuplas de v y r cumplan ciertas restricciones.
- `GROUP BY` donde se agrupa por un atributo.
- `HAVING` donde se usa la función de agregación `SUM` para filtrar el resultado de manera que solo se muestran los grupos formados cuya suma sea mayor o igual que cuatro.

En la Figura 4.6 se puede ver el árbol de dependencias sintácticas de la vista v4.

Resultado obtenido:

r		
4	8	0
0	0	0

s	
0	8
1	0

v4
8

Para que la consulta que define la vista proporcione al menos una tupla válida hay que tener varias cosas en cuenta: En la vista v, se pide que el valor de v.b = s.b = 8. En la condición del `WHERE` de la consulta v4 se pide, además, que v.b = r.b, por lo tanto esta columna también debe tomar valor 8. Y por último una vez agrupadas las tuplas de la tabla r tales que su r.b = 8, la suma de los valores de la columna r.a debe ser igual a 4. Como se puede observar, todas estas

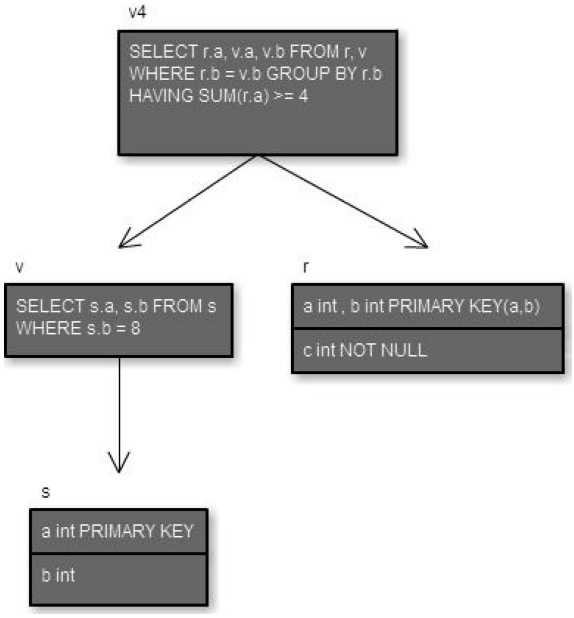

Figura 4.6: Esquema en árbol de vista de v4

condiciones se cumplen en el caso de prueba generado.

Salida DES:

```
DES-SQL> /test_case v4 add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|8|
|1|0|
-----------
r
-----------
|4|8|0|
|0|0|0|
```

Tras ejecutar la consulta v4 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v4

answer(v4.a1:number(integer)) ->
{
answer(8)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

A medida que se hacen más complejos los ejemplos, las fórmulas se van haciendo más largas. En este ejemplo hemos decidido omitir la fórmula ya que es muy grande y no aporta claridad al ejemplo.

4.7 EJEMPLO 7:Vista con cláusulas GROUP BY y HAVING y función de agregación COUNT

Sea la vista v5 definida como sigue:

```
CREATE OR REPLACE VIEW v5(a1) AS
SELECT r.b
FROM r, v
WHERE r.b = v.b
GROUP BY r.b
HAVING COUNT(r.a) =6;
```

En este ejemplo la instancia de la base de datos tiene que tener 3 registros en cada tabla involucrada para que, después de hacer el producto cartesiano, al agrupar por r.b, el número de filas de r.a contenidas en el grupo sea igual a 6. En la Figura 4.7 se puede ver el árbol de dependencias sintácticas de la vista v5.

Resultado obtenido:

r		
0	0	0
0	8	0
1	8	0

s	
0	8
1	8
2	8

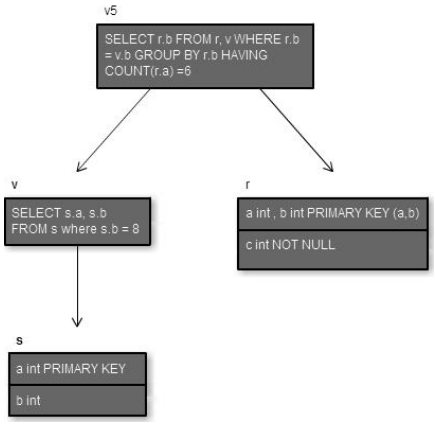

Figura 4.7: Esquema en árbol de vista de v5

v5
8

Salida DES:

```
DES-SQL> /test_case v5 add
Prueba con test case de tamaño 2
Prueba con test case de tamaño 3
Info:  Test case sobre enteros:
-----------
s
-----------
|0|8|
|1|8|
|2|8|
-----------
r
-----------
|0|0|0|
|0|8|0|
|1|8|0|
```

Tras ejecutar la consulta v5 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v5

answer(v5.a1:number(integer)) ->
{
answer(8)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

Por la misma razón que en el ejemplo anterior, hemos decidido omitir la fórmula.

4.8 EJEMPLO 8: Vista que hace la unión de dos consultas

Sea la vista v6 definida como sigue:

```
CREATE OR REPLACE VIEW v6(a1,a2) AS
SELECT r.a, r.b
FROM r
UNION
SELECT *
FROM v;
```

La vista v6 esta construida sobre la unión de dos consultas sencillas usando el operador `UNION`. En la Figura 4.8 se puede ver el árbol de dependencias sintácticas de la vista v6.

Resultado obtenido:

r		
0	0	0
0	1	0

s	
0	0
1	0

v6	
0	0
0	1

En esta consulta cabe destacar que para que la unión de dos consultas tenga valores sólo es necesario que una de las dos los tenga. Ya que la unión coge todos los valores tanto de una como de la otra.

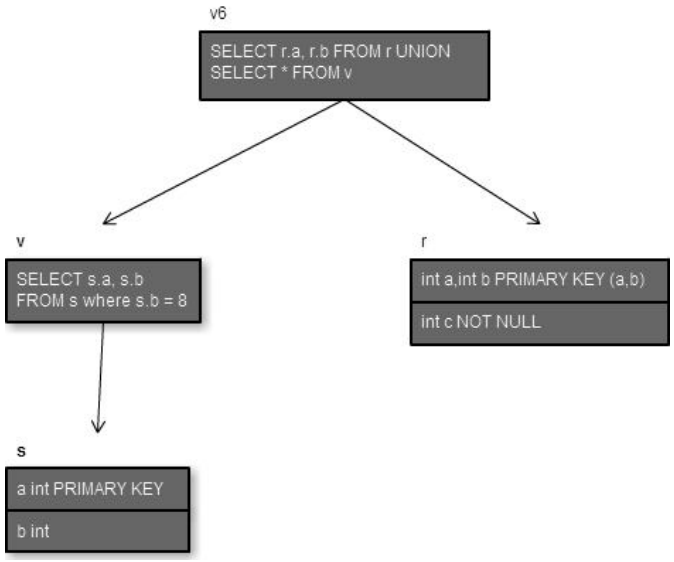

Figura 4.8: Esquema en árbol de vista de v6

Salida DES:

```
DES-SQL> /test_case v6 add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|0|
|1|0|
-----------
r
-----------
|0|0|0|
|0|1|0|
```

Tras ejecutar la consulta v6 con la instancia generada, el resultado obtenido contiene dos tuplas, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v6

answer(v6.a1:number(integer),v6.a2:number(integer)) ->
{
answer(0,0),
answer(0,1)
}
Info:  2 tuples computed.
```

Fórmulas obtenidas:

$((((0 \neq 0) \vee (0 \neq 1)) \wedge (0 \neq null)) \wedge ((1 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge (0 \neq null)))))$

4.9 EJEMPLO 9: Vista que hace la intersección de dos consultas

Sea la vista v7 definida como sigue:

```
CREATE OR REPLACE VIEW v7(a1,a2) AS
SELECT r.a, r.b
FROM r
INTERSECT
SELECT * FROM v;
```

La vista v7 esta construida sobre la intersección de dos consultas sencillas usando el operador `INTERSECT`. En la Figura 4.9 se puede ver el árbol de dependencias sintácticas de la vista v7.

Resultado obtenido:

r		
1	8	0
0	0	0

s	
0	0
1	8

v7	
1	8

En esta consulta cabe destacar que para que la intersección de dos consultas tenga un valor como mínimo, tiene que haber al menos una tupla igual en las dos consultas. En este caso la $(1, 8)$.

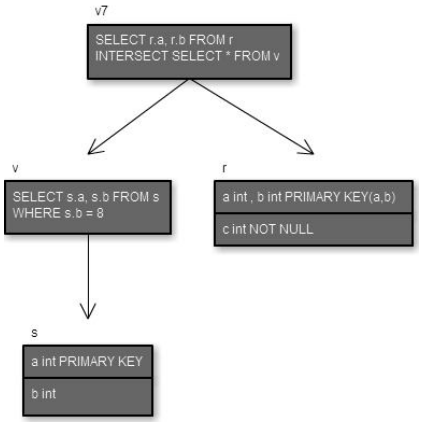

Figura 4.9: Esquema en árbol de vista de v7

Salida DES:

```
DES-SQL> /test_case v7 add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|0|
|1|8|
-----------
r
-----------
|1|8|0|
|0|0|0|
```

Tras ejecutar la consulta v7 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v7

answer(v7.a1:number(integer),v7.a2:number(integer)) ->
{
answer(1,8)
}
Info:  1 tuple computed.
```

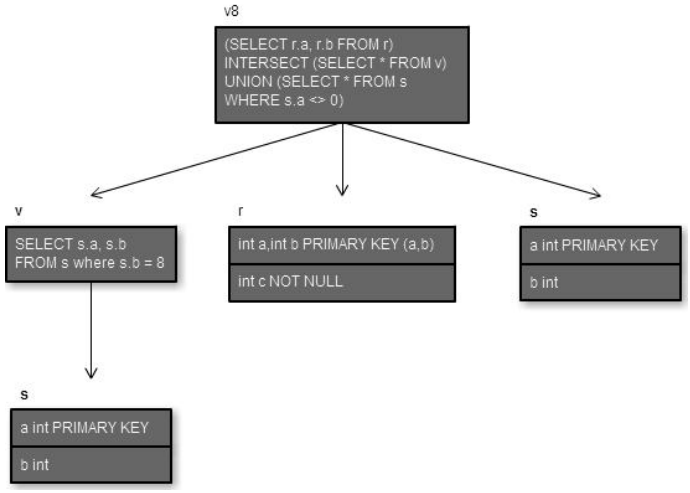

Figura 4.10: Esquema en árbol de vista de v8

Fórmulas obtenidas:

$(((((((1 \neq 0) \vee (8 \neq 0)) \wedge (0 \neq null)) \wedge ((0 \neq 1) \wedge (0 = 8))) \wedge ((1 = 0) \wedge (8 = 0))) \vee (((((1 \neq 0) \vee (8 \neq 0)) \wedge (0 \neq null)) \wedge ((1 \neq 0) \wedge (8 = 8))) \wedge ((1 = 1) \wedge (8 = 8)))) \wedge ((8 \neq null) \wedge ((0 \neq null) \wedge ((1 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((0 \neq null) \wedge (1 \neq null)))))))))$

4.10 EJEMPLO 10: Vista que anida unión e intersección

Sea la vista v8 definida como sigue:

```
CREATE OR REPLACE VIEW v8(a1,a2) AS
(SELECT r.a, r.b FROM r)
INTERSECT
(SELECT * FROM v)
UNION
(SELECT * FROM s WHERE s.a <> 0);
```

La vista v8 esta definida como la intersección de una unión, de forma que se muestra la posibilidad de anidar estas dos cláusulas. En la Figura 4.10 se puede ver el árbol de dependencias sintácticas de la vista v8.

Resultado obtenido:

r		
1	0	0
0	0	0

s	
0	0
1	0

v8	
1	0

Salida DES:

```
DES-SQL> /test_case v8
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|0|
|1|0|
-----------
r
-----------
|1|0|0|
|0|0|0|
```

Tras ejecutar la consulta v8 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v8

answer(v8.a1:number(integer),v8.a2:number(integer)) ->
{
answer(1,0)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

En este ejemplo hemos decido omitir la fórmula ya que es muy grande y no aporta claridad al ejemplo.

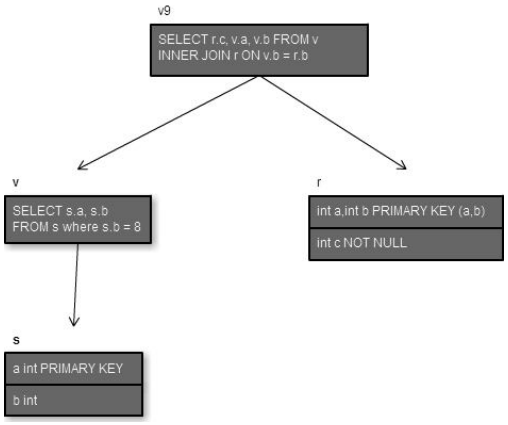

Figura 4.11: Esquema en árbol de vista de v9

4.11 EJEMPLO 11: Vista con cláusula INNER JOIN

Sea la vista v9 definida como sigue:

```
CREATE OR REPLACE VIEW v9(a1,a2, a3) AS
SELECT r.c, v.a, v.b
FROM v
INNER JOIN r ON v.b = r.b;
```

La vista v9 está definida por una consulta simple que realiza el producto cartesiano entre la vista v y la tabla r por medio de la cláusula `INNER JOIN`. En la Figura 4.11 se puede ver el árbol de dependencias sintácticas de la vista v9.

Resultado obtenido:

r		
0	8	0
0	0	0

s	
0	8
1	0

v8		
0	0	8

Salida DES:

```
DES-SQL> /test_case v9 add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|8|
|1|0|
-----------
r
-----------
|0|8|0|
|0|0|0|
```

Tras ejecutar la consulta v9 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v9

answer(v9.a1:number(integer),v9.a2:number(integer),v9.a3:number(integer))
->
{
answer(0,0,8)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

$(((((0 \neq 1) \wedge (8 = 8)) \wedge (((0 \neq 0) \vee (8 \neq 0)) \wedge (0 \neq null))) \wedge (8 = 8)) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((1 \neq null) \wedge (0 \neq null)))))))$

4.12 EJEMPLO 12: Vista con cláusula INNER JOIN anidados

Sea la vista v10 definida como sigue:

```
CREATE OR REPLACE VIEW v10(a1,a2, a3, a4) AS
SELECT r.c, v.a, v.b, s.a
FROM v INNER JOIN r ON v.b = r.b INNER JOIN s ON v.a = s.a;
```

La vista v10 está definida por una consulta simple que realiza el producto cartesiano entre la vista v y las tablas r y s por anidando varias cláusulas `INNER JOIN`. En la Figura 4.12 se puede ver el

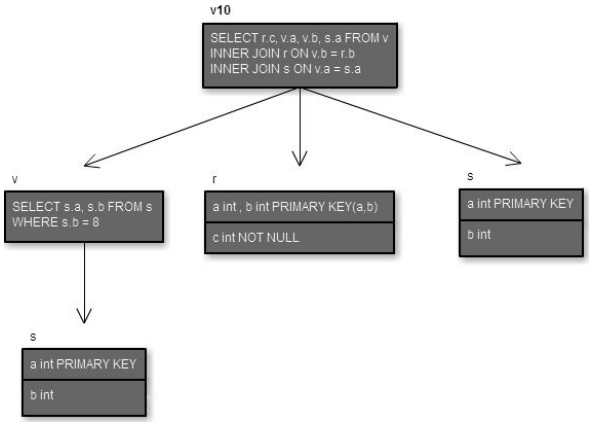

Figura 4.12: Esquema en árbol de vista de v10

árbol de dependencias sintácticas de la vista v10.

Resultado obtenido:

r		
0	8	0
0	0	0

s	
0	8
1	0

v10			
0	0	8	0

Salida DES:

```
DES-SQL> /test_case v10 add
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|8|
|1|0|
-----------
r
-----------
```

```
|0|8|0|
|0|0|0|
```

Tras ejecutar la consulta v10 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v10

answer(v10.a1:number(integer),v10.a2:number(integer),
v10.a3:number(integer),v10.a4:number(integer)) ->
{
answer(0,0,8,0)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

$((((((0 \neq 0) \vee (8 \neq 0)) \wedge (0 \neq null)) \wedge (((0 \neq 1) \wedge (8 = 8)) \wedge (0 \neq 1))) \wedge ((0 = 0) \wedge (8 = 8))) \wedge ((8 \neq null) \wedge ((1 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((0 \neq null) \wedge (0 \neq null)))))))$

4.13 EJEMPLO 13: Vista con cláusula DISTINCT

Sea la vista v11 definida como sigue:

```
CREATE OR REPLACE VIEW v11(a) AS
SELECT DISTINCT w.a
FROM w
WHERE w.a = 4;
```

La vista v11 esta definida sobre una consulta en la que aparece la cláusula `DISTINCT` como método para eliminar las repeticiones. En la Figura 4.13 se puede ver el árbol de dependencias sintácticas de la vista v11.

Resultado obtenido:

s	
0	8
1	0

r		
4	8	0
0	0	0

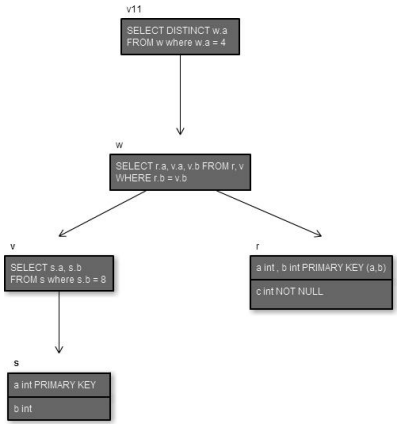

Figura 4.13: Esquema en árbol de vista de v11

v11
4

Salida DES:

```
DES-SQL> /test_case v11
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|8|
|1|0|
-----------
r
-----------
|4|8|0|
|0|0|0|
```

Tras ejecutar la consulta v11 con la instancia generada, el resultado obtenido contiene una tupla, por lo tanto se trata de un caso de prueba válido.

```
DES-SQL> select * from v11
```

```
answer(v11.a:number(integer)) ->
{
answer(4)
}
Info:  1 tuple computed.
```

Fórmulas obtenidas:

$(((((((((4 \neq 0) \vee (8 \neq 0)) \wedge (0 \neq null)) \wedge ((0 \neq 1) \wedge (8 = 8))) \wedge (8 = 8)) \wedge (4 = 4)) \wedge ((\neg((((((0 \neq 4) \vee (0 \neq 8)) \wedge (0 \neq null)) \wedge ((0 \neq 1) \wedge (8 = 8))) \wedge (0 = 8)) \wedge (0 = 4))((((((((0 \neq 4) \vee (0 \neq 8)) \wedge (0 \neq null)) \wedge ((0 \neq 1) \wedge (8 = 8))) \wedge (0 = 8)) \wedge (0 = 4))) \vee (4 \neq 0)) \wedge (\neg((((((0 \neq 4) \vee (0 \neq 8)) \wedge (0 \neq null)) \wedge ((1 \neq 0) \wedge (0 = 8))) \wedge (0 = 0)) \wedge (0 = 4))((((((((0 \neq 4) \vee (0 \neq 8)) \wedge (0 \neq null)) \wedge ((1 \neq 0) \wedge (0 = 8))) \wedge (0 = 0)) \wedge (0 = 4))) \vee (4 \neq 0)))) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((1 \neq null) \wedge ((0 \neq null) \wedge ((0 \neq null) \wedge ((8 \neq null) \wedge ((0 \neq null) \wedge (4 \neq null)))))))))$

4.14 EJEMPLO 14: Vista con nulos que involucra una clave primaria

Sea la vista v14 definida como sigue:

```
CREATE OR REPLACE VIEW v14(a) AS
SELECT DISTINCT t.a
FROM t
WHERE t.a = null;
```

La vista v14 esta formada por una consulta muy sencilla que solamente involucra una tabla. El campo a de esta tabla t forma parte de su clave primaria, por lo tanto no es posible obtener un registro para esta vista ya que en la condición del WHERE se requiere que t.a sea nulo y esto es incompatible con la definición de la tabla. En la Figura 4.14 se puede ver el árbol de dependencias sintácticas de la vista v14.

Salida DES:

```
DES-SQL> /test_case v14
Prueba con test case de tamaño 2
Prueba con test case de tamaño 3
Prueba con test case de tamaño 4
```

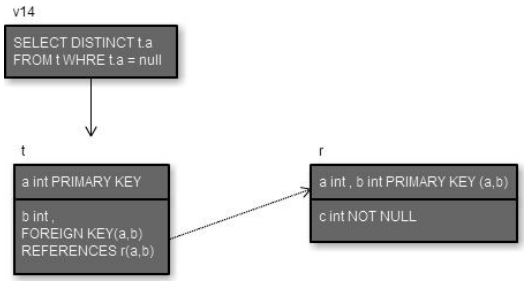

Figura 4.14: Esquema en árbol de vista de v14

```
Prueba con test case de tamaño 5
Prueba con test case de tamaño 6
Prueba con test case de tamaño 7
Info:  No se puede generar el test case.
```

El sistema intenta obtener el caso de prueba con hasta 7 registros que es el límite máximo. Una vez que supera este máximo para y muestra por pantalla que no se pudo generar una instancia de la base de datos válida para esa vista.

4.15 EJEMPLO 15: Vista con nulos

Sea la vista v15 definida como sigue:

```
CREATE OR REPLACE VIEW v15(a) AS
SELECT DISTINCT s.a
FROM s
where s.b = null;
```

La vista v15 esta formada por una consulta muy sencilla que solamente involucra una tabla. El único requisito que se establece en la consulta viene de la cláusula `WHERE`, en la que se establece que el campo s.b tiene que ser igual a *null*. En la Figura 4.15 se puede ver el árbol de dependencias sintácticas de la vista v15.

Resultado obtenido:

s	
0	null
1	0

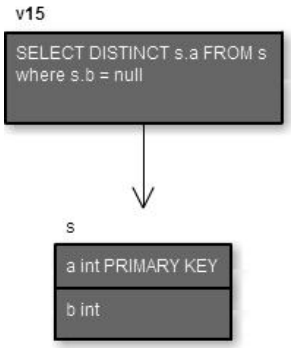

Figura 4.15: Esquema en árbol de vista de v15

v15
0

Salida DES:

```
DES-SQL> /test_case v15
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|null|
|1|0|
```

Fórmulas obtenidas:

$((((0 \neq 1) \wedge (0 = null)) \wedge (\neg((1 \neq 0) \wedge (0 = null))(((1 \neq 0) \wedge (0 = null))) \vee (0 \neq 1))) \wedge ((1 \neq null) \wedge (0 \neq null)))$

Esta es una consulta sencilla en la que se muestra que es posible generar casos de prueba con valores nulos en caso de que así lo especifique la consulta tratada.

4.16 EJEMPLO 16: Vista con nulos y operador lógico OR

Sea la vista v16 definida como sigue:

```
CREATE OR REPLACE VIEW v16(a) AS SELECT DISTINCT s.a
FROM s
```

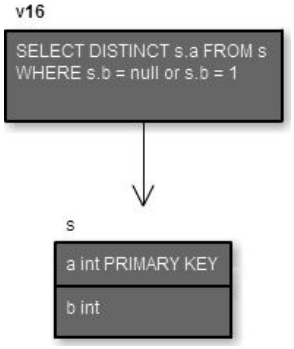

Figura 4.16: Esquema en árbol de vista de v16

```
where s.b = null or s.b = 1;
```

La vista v16 esta definida por una consulta sencilla. En esta consulta, en la cláusula `WHERE`, se esta pidiendo que se cumpla una disyunción de condiciones, es decir, que se cumpla una de ellas o la otra. En este caso la primera condición pide que s.b = null y la segunda que s.b = 1. El caso de prueba generado para esta consulta podría tomar cualquiera de estos dos valores, seleccionando el que s.b = 1, puesto que a la hora de ejecutar el resolutor de restricciones, éste siempre toma el valor más pequeño de entre todas sus posibilidades. Entonces, la variable que almacena el estado de s.b toma el valor 0, teniendo en cuenta que solo puede elegir entre 0 y 1, lo que significará que la variable en cuestión no toma el valor nulo. En la Figura 4.16 se puede ver el árbol de dependencias sintácticas de la vista v16.

Resultado obtenido:

s	
0	1
1	0

v16
0

Salida DES:

```
DES-SQL> /test_case v16
Prueba con test case de tamaño 2
Info:  Test case sobre enteros:
-----------
s
-----------
|0|1|
|1|0|
```

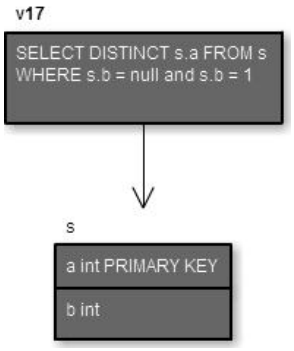

Figura 4.17: Esquema en árbol de vista de v17

Fórmulas obtenidas:

$((((0 \neq 1) \wedge ((1 = null) \vee (1 = 1))) \wedge (\neg((1 \neq 0) \wedge ((0 = null) \vee (0 = 1)))(((1 \neq 0) \wedge ((0 = null) \vee (0 = 1)))) \vee (0 \neq 1))) \wedge ((0 \neq null) \wedge ((1 \neq null) \wedge ((1 \neq null) \wedge (0 \neq null)))))$

4.17 EJEMPLO 17: Vista con nulos y operador lógico AND

Sea la vista v16 definida como sigue:

```
CREATE OR REPLACE VIEW v17(a) AS SELECT DISTINCT s.a
FROM s
where s.b = null and s.b = 1;
```

La vista v17 esta definida sobre una consulta sencilla que contiene en su cláusula `WHERE` una condición `AND` insatisfactible. En la Figura 4.17 se puede ver el árbol de dependencias sintácticas de la vista v17.

Salida DES:

```
DES-SQL> /test_case v17
Prueba con test case de tamaño 2
Prueba con test case de tamaño 3
Prueba con test case de tamaño 4
Prueba con test case de tamaño 5
Prueba con test case de tamaño 6
```

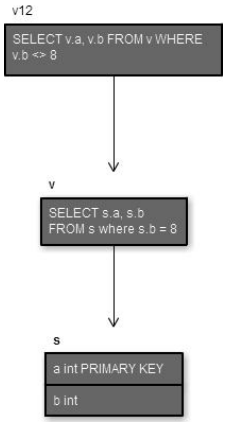

Figura 4.18: Esquema en árbol de vista de v12

```
Prueba con test case de tamaño 7
Info:  No se puede generar el test case.
```

No se puede generar el caso de prueba dado que no es posible que un campo tenga valor null y 1 a la vez.

4.18 EJEMPLO 18: Vista con fórmulas insatisfactibles

Sea la vista v12 definida como sigue:

```
CREATE OR REPLACE VIEW v12(a1,a2) AS
SELECT v.a, v.b
FROM v
WHERE v.b <> 8;
```

En la Figura 4.18 se puede ver el árbol de dependencias sintácticas de la vista v12.

Salida DES:

```
DES-SQL> /test_case v12
Prueba con test case de tamaño 2
Prueba con test case de tamaño 3
Prueba con test case de tamaño 4
Prueba con test case de tamaño 5
Prueba con test case de tamaño 6
```

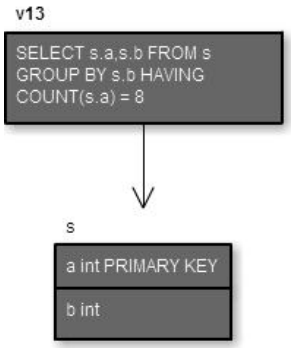

Figura 4.19: Esquema en árbol de vista de v13

```
Prueba con test case de tamaño 7
Info:  No se puede generar el test case.
```

En este caso hay una contradicción en la consulta, dado que está pidiendo que r.b sea distinto de 8 y sin embargo, la vista v tiene como condición que r.b sea igual a 8. Por tanto, no puede generar el caso de prueba.

4.19 EJEMPLO 19: Vista con fórmulas insatisfactibles

Sea la vista v13 definida como sigue:

```
CREATE OR REPLACE VIEW v13(a1,a2) AS SELECT s.a,s.b
FROM s
GROUP BY s.b
HAVING COUNT(s.a) = 8;
```

En la Figura 4.19 se puede ver el árbol de dependencias sintácticas de la vista v13.

Salida DES:

```
DES-SQL> /test_case v13
Prueba con test case de tamaño 2
Prueba con test case de tamaño 3
Prueba con test case de tamaño 4
Prueba con test case de tamaño 5
Prueba con test case de tamaño 6
Prueba con test case de tamaño 7
Info:  No se puede generar el test case.
```

En este caso no puede generar el caso de prueba dado que el tamaño máximo del caso de prueba es 7 (/tc_size 2 7) y la consulta requiere que como mínimo haya 8 registros.

Capítulo 5

Conclusiones y trabajo futuro

En esta sección describimos las características de la aplicación, así como las mejoras que se podrían realizar en un futuro.

5.1 Conclusiones

Nuestra principal aportación a este sistema, es la generación de las fórmulas que tiene que cumplir una vista para ser un caso de prueba positivo.

Las fórmulas generadas pasan por un proceso de optimización por el que se transforman en fórmulas más claras y eficientes. Al crearlas, existen muchas expresiones redundantes que posteriormente se simplifican, por ejemplo, las expresiones $and(X, true)$ y $or(X, false)$ se simplifican a X. Dicha transformación a fórmulas más simples facilita la comprensión de las mismas y hace que el sistema consiga mayor velocidad a la hora de resolver las restricciones.

Además, al existir esta separación entre generación de fórmulas y lanzamiento de restricciones, el sistema adquiere una mayor modularidad, lo que aporta bastantes ventajas, como la posibilidad de modificar un módulo sin que afecte a otras partes del código o la facilidad de detección y aislamiento de errores.

Por otro lado, hemos extendido la sintaxis permitida añadiendo el tratamiento de la cláusula `DISTINCT`, el `INNER JOIN` permitiendo anidaciones, el uso de los operadores `UNION` e `INTERSECT`, también permitiendo la anidación de ambos, y el tratamiento de valores nulos.

Este proyecto es un paso importante para la generación de casos de prueba que verifican SQLFpc.

5.2 Trabajo futuro

A partir del trabajo realizado se han identificado una serie de necesidades y posibles mejoras de la herramienta. A continuación mostramos una clasificación:

- *Ampliación de la sintaxis SQL:*

  Sería interesante ampliar la funcionalidad del sistema, permitiendo introducir en la definición de las vistas, para las que se quiere generar el caso de prueba, más sintaxis SQL. Algunos ejemplo podrían ser:

  - `OUTER JOINS`: Un hito importante en el desarrollo de nuestro proyecto ha sido el tratar los valores nulos, así añadir las cláusulas `LEFT OUTER JOIN, RIGHT OUTER JOIN, FULL OUTER JOIN` que manejan estos valores no debería plantear ningún problema.
  - `EXISTS, BETWEEN, IN, ANY, ALL`: En nuestro programa está implementada la funcionalidad de realizar subconsultas en la sección `FROM` (reescribiéndolas como vistas SQL), pero no en el `WHERE`. Incluir el tratamiento de estas cláusulas sería beneficioso para extender la utilidad del sistema ya que se usan comúnmente.
  - **Funciones de agregación**: En nuestro programa están implementadas algunas funciones de agregación (ver Sección 2.3) para la sección `HAVING` de las consultas. Añadir las faltantes, véase `MIN, MAX, AVG`, y permitir su uso también en la sección `SELECT` sería otro hecho importante.

- *Generación de más casos de prueba:*

  Como explicamos en la Sección 1.3, esta herramienta busca un caso de prueba positivo para una vista. Con esta prueba únicamente no podemos asegurar que la consulta generadora de dicha vista sea correcta, sino que es necesario generar más casos de prueba. La generación de las variantes de la consulta original que representan las reglas definidas por el criterio de cobertura SQLFpc y la unión con nuestro proyecto para que las trate y genere sus correspondientes casos de prueba, construyendo así el conjunto completo de casos de prueba que es necesario para ayudar al usuario a encontrar posibles errores en la consulta inicial, quedan como trabajo futuro.

Apéndice A

Manual de usuario

A continuación se describen los pasos necesarios para ejecutar la aplicación y algunos problemas frecuentes que pueden surgir junto con sus posibles soluciones.

A.1 Pasos necesarios para ejecutar la aplicación

1. **Obtener DES.**
   Es necesario obtener el directorio con todos los archivos indispensables para la ejecución de DES

2. **Descargar e instalar SICStus Prolog.**
   Es posible descargarlo desde su página web oficial: `http://sicstus.sics.se/`. Se trata de software propietario por lo que se necesita una licencia válida para poder utilizarlo.

   Una vez instalado, para mayor comodidad, se puede cambiar el directorio de trabajo por defecto a la carpeta donde se encuentren los archivos de DES de la siguiente forma: hacer clic derecho sobre el icono de SICStus Prolog, pulsar en propiedades y posteriormente colocar la ruta correspondiente en la opción *iniciar en:*

3. **Consultar el archivo DES.pl.**
   DES.pl es el archivo principal de DES, se encarga de incluir todo lo necesario para la correcta ejecución del sistema.

   Es posible consultarlo mediante las opciones del menú (File->Consult) o bien, en el caso de que se tenga incluido dicho archivo en el directorio por defecto, directamente escribiendo por consola el comando "[des].". Posteriormente, deberá aparecer en pantalla lo siguiente: `DES>`

4. **Introducir en el sistema las tablas y vistas.**
   Antes de introducir las tablas y vistas que formarán parte de la base de datos, es necesario cambiar el modo de lenguaje a SQL, esto se indica con el comando `/sql` apareciendo después en pantalla: `DES_SQL>`. Se pueden introducir las tablas y vistas directamente por consola, o bien cargarlas mediante un fichero utilizando el comando `/p nombredelfichero`.

5. **Obtener casos de prueba.**
   Por último, mediante `/test_case nombrevista`, se intentará generar una instancia de la base de datos con las tuplas necesarias para encontrar un caso de prueba positivo (ver Figura

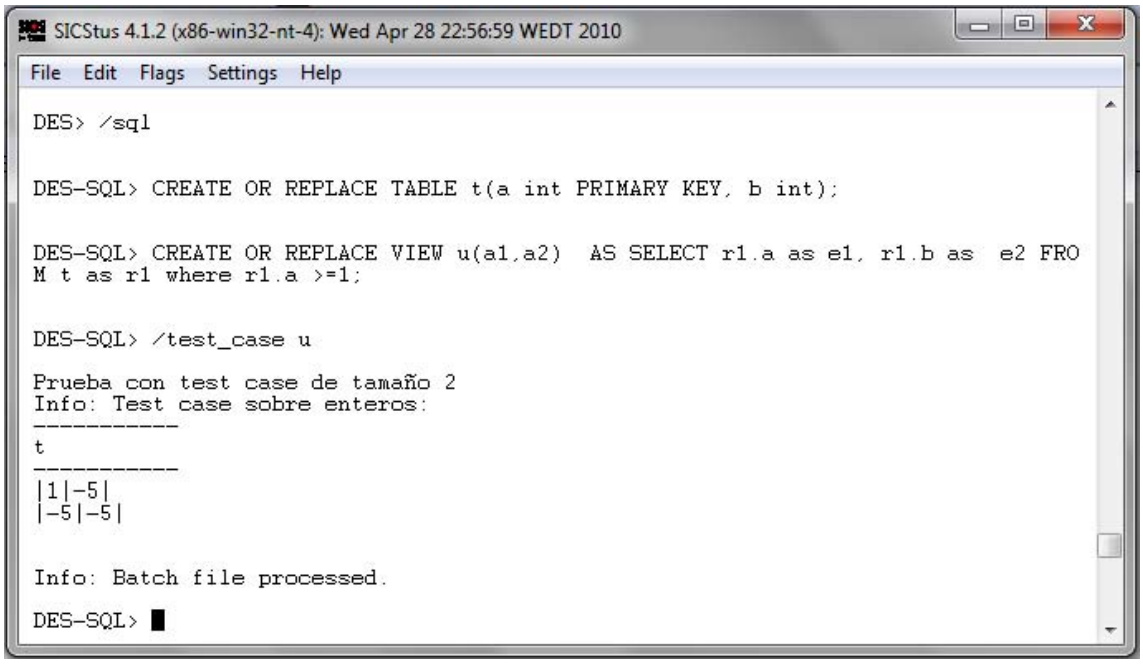

Figura A.1: Pasos necesarios para ejecutar la aplicación

A.1) o, si no se puede generar, mostrará un mensaje de error en pantalla para informar de este hecho.

A.2 Problemas frecuentes

Si al lanzar una prueba sobre una determinada consulta no se obtiene ningún resultado, es indicio de que la consulta es errónea y hará falta revisarla para corregirla. Sin embargo, existen otros parámetros propios del sistema generador de casos de prueba que se deben tener en cuenta, ya que pueden provocar que una consulta sea considerada como errónea cuando en realidad no lo sería si se ajustaran:

- **Tamaño mínimo y máximo del caso de prueba (tc_size)**: Indica cuántos registros va a intentar generar el sistema como mínimo y máximo respectivamente. Es bastante importante que este parámetro esté ajustado al número de registros necesarios para realizar la consulta, dado que puede haber consultas que requieran generar un mayor número de registros del límite máximo guardado en el sistema.

  Para modificarlo, hay que hacer uso del comando `/tc_size`. Por ejemplo, `/tc_size 5 7` indica al sistema que se desea obtener un caso de prueba de tamaño mínimo 5 y máximo 7.

- **Dominio del caso de prueba (tc_domain)**: Establece el rango de valores que pueden tomar cada uno de los campos de todas las tablas y vistas pertenecientes a la base de datos. También es necesario que los valores del dominio estén ajustados a los valores que se exigen

en la consulta dentro de la cláusula where y having.

Para modificarlo, hay que hacer uso del comando `/tc_domain`. Por ejemplo, `/tc_domain -5 10` indica al sistema que los casos de prueba tienen que tomar como valor números enteros comprendidos entre -5 y 10.

Bibliografía

[1] C. Binnig, D. Kossmann, and E. Lo. Reverse query processing. 2007. Available from: http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.84.9131.

[2] C. Binnig, D. Kossmann, and E. Lo. Towards automatic test database generation. *IEEE Data Eng. Bull.*, 31(1):28–35, 2008. Available from: http://sites.computer.org/debull/A08mar/binnig.pdf.

[3] C. Binnig, D. Kossmann, E. Lo, M. Nunkesser, and T. Ozsu. Qagen: Generating query-aware test databases. 2007. Available from: http://e-collection.library.ethz.ch/eserv/eth:4877/eth-4877-01.pdf.

[4] R. Caballero, Y. García-Ruiz, and F. Sáenz-Pérez. Applying Constraint Logic Programming to SQL Test Case Generation. In *Proc. International Symposium on Functional and Logic Programming (FLOPS'10)*, volume 6009/2010 of *Lecture Notes in Computer Science*, pages 191–206. Springer, 2010. Available from: http://www.fdi.ucm.es/profesor/fernan/FSP/CGS10a.pdf.

[5] M. Carlsson. Sicstus prolog user's manual version 4.2.3. Technical report. Available from: http://sicstus.sics.se/.

[6] F. Sáenz-Pérez. Datalog educational system v3.0 user's manual. Technical report, Faculty of Computer Science, UCM. Available from: http://des.sourceforge.net/.

[7] J. Tuya, M. J. Suárez-Cabalan, and C. de la Riva. Full predicate coverage for testing sql database queries. 2010. Available from: http://giis.uniovi.es/testing/papers/stvr-2010-sqlfpc.pdf.

[8] Wikipedia. Definición de base de datos. Available from: http://es.wikipedia.org/wiki/Base_de_datos.

[9] Wikipedia. Definición de información. Available from: http://es.wikipedia.org/wiki/Informaci\unhbox\voidb@x\bgroup\let\unhbox\voidb@x\setbox\@tempboxa\hbox{o\global\mathchardef\accent@spacefactor\spacefactor}\accent19o\egroup\spacefactor\accent@spacefactorn.