Tesis de Licenciatura

Visualización de Estrategias de testing para bases de datos activas

Juan Pablo PORTAL

Universidad de Buenos Aires

(UBA)

Directores:

Prof. Daniel YANKELEVICH &
Prof. Martina MARRE

Indice

1 Introducción

La ingeniería del software es una disciplina relativamente joven. Durante las últimas décadas el uso de aplicaciones crecientes en complejidad y tamaño ha aumentado considerablemente. Consecuentemente, el proceso de desarrollo de aplicaciones se ha ido industrializando y adquiriendo similitudes con el área de ingeniería. La ingeniería del software agrupa un conjunto de técnicas aplicables a la construcción y el mantenimiento de aplicaciones informáticas. Estas técnicas incluyen actividades diversas como la administración, elaboración de presupuestos, modelización, análisis, especificación, diseño, implementación, testing y mantenimiento.

En esta tesis estudiaremos una de las técnicas enumeradas anteriormente aplicada a un universo de aplicaciones posibles: testing de base de datos activas. La tesis presenta dos ejes fundamentales: uno, la *aplicación desarrollada* (el "ADB_Tester") y el otro, el teórico que brinda un marco formal y nos permite *comprender* desde un acercamiento científico y metodológico la información obtenida. Para estudiar el testing de base de datos activas, deberemos estudiar antes el testing del software en general por un lado y las bases de datos activas por el otro.

Testing de Software

La validación y verificación de un sistema de software es esencial durante todo el proceso de desarrollo del mismo. La técnica predominante de validación y verificación [ABC82] usada durante la producción de software es el *testing*. El testing es esencial para encontrar errores, mejorar la calidad y *dar confianza en la calidad de un programa*, constituyéndose así en un paso fundamental del control de calidad de un desarrollo informático.

Para un input dado, el testing consiste en determinar el comportamiento *esperado* del sistema con ese input, ejecutar el programa con dicho input y obtener el comportamiento *real* del sistema, y finalmente comparar el comportamiento real con el esperado. Si coinciden, el sistema funciona bien para dicho input. En caso contrario, se encontró un error en el sistema.

La selección de un subconjunto representativo de los inputs posibles puede hacerse usando diferentes *estrategias* o *criterios de selección de casos de test*. Una estrategia de selección de casos de testing trata de agrupar elementos del dominio de inputs que tienen un comportamiento *similar* en una misma clase, para luego seleccionar datos de test de cada clase de equivalencia. La idea es que para los propósitos del testing todos los elementos de una clase de equivalencia son esencialmente el mismo. Es decir, se asume que cualquier elemento de una clase va a evidenciar el error tanto como cualquier otro elemento de la misma clase.

Bases de Datos Activas [DHW94]

Las bases de datos juegan un rol cada vez más importante en el desarrollo de sistemas de información. Muchos de estos sistemas comienzan a describirse como clientes que interactúan con un servidor de base de datos que centraliza los datos y almacena la información, conformando así la médula del sistema. Recientemente, se ha trabajado en la dirección de extender las tradicionales bases de datos relacionales con facilidades adicionales. Este trabajo provino tanto de grupos de investigación, como de grupos de estándares (SQL 92 y ODMG) y

de la industria. En particular, la mayor parte de los manejadores de bases de datos relacionales incorporaron o planean incorporar en el corto plazo funcionalidades adicionales que permiten la ejecución de reglas (triggers) que se ejecutan frente a determinados eventos, la definición de tipos abstractos de datos o la incorporación de objetos en la base.

Las bases de datos activas son bases de datos que *incorporan componentes dinámicas dentro de la base*. En general, se asume que dichas componentes tienen la forma de reglas que se disparan frente a eventos y ejecutan una porción de código. Dicho código puede habilitar nuevos eventos, que a su vez disparen nuevas reglas, aumentando la complejidad del sistema.

Testing de Bases de Datos Activas

En el marco de esta tesis, intentaremos aplicar los conocimientos del testing a la problemática de las Bases de Datos Activas. Todavía la ingeniería del software no avanzó demasiado en esta dirección; sin embargo la creciente utilización de la "actividad" de las bases de datos obliga a hacerlo. Las reglas de una base de datos forman parte de una aplicación y como tal deben ser analizadas, testeadas, mantenidas, etc.

En el área de testing contamos con el aporte de [Bal97].

Tomando el trabajo anteriormente mencionado como punto de partida, hemos realizado este trabajo.

La herramienta desarrollada en el marco de esta tesis permite entre otras cosas conocer la estructura interna de la aplicación a testear y comprobar la "efectividad" del testing una vez realizado.

Objetivos buscados en esta tesis:

Investigar la automatización de las tareas de testing estructural para base de datos activas, desarrollando las bases para la generación de herramientas semiautomáticas de testing.

Construir una herramienta que facilite el Testing de Bases de Datos Activas, la denominaremos ADB_Tester.

Validar, mediante experimentos, diferentes estrategias de testing para base de datos activas, comparándolas y obteniendo conclusiones empíricas en un marco riguroso.

Estructura de la tesis

En el capítulo siguiente veremos los fundamentos teóricos requeridos para la comprensión de la tesis y para la construcción de la herramienta ADB_Tester. Siendo ésta una parte fundamental de la tesis, haremos especial hincapié en los temas referidos a Oracle (estructura y propiedades de los triggers, y la sintáxis del PL/SQL, el lenguaje con el que se programan los triggers de Oracle).

En el capítulo 3, presentaremos la metodología propuesta para realizar el testing de una base de datos activa. Se conocerá en este capítulo, cuáles son las herramientas desarrolladas durante este trabajo. Dichas herramientas serán presentadas con un mayor detalle funcional en el capítulo 4 y se podrán leer sus especificaciones técnicas y manuales en los diferentes apéndices.

En el capítulo 5 presentaremos tres casos de estudio. El primero de ellos es un caso real de utilización de elementos activos en una base de datos. El interés de este caso radica en que se lo puede considerar como un ejemplo estándar de la utilización de reglas dentro de los usos

y costumbres de la industria informática argentina de nuestros días. El segundo ejemplo es un caso teórico que intenta mostrar lo complicadas que pueden ser las relaciones entre las reglas de una base de datos y la utilidad de una herramienta como el ADB_Tester.

Finalmente, en el capítulo 6 presentamos conclusiones de este trabajo y posibles vías de estudio para trabajos futuros.

2 Background - Fundamentos

Presentaremos en este capítulo cuatro áreas de investigación que debieron ser estudiadas para la realización de este trabajo. En primer lugar haremos una rápida presentación de los conceptos fundamentales de las bases de datos activas, tema que será profundizado en la sección dedicada a Oracle. Luego brindaremos las nociones indispensables sobre DBGraphs, cubrimientos y testing.

2.1 Bases de datos activas – Modelo teórico

El modelo elegido como base en la tesis de licenciatura [Bal97] es el de Starburst [WF90]. En cambio, al ser ésta una tesis implementativa el modelo real diferirá en aspectos que veremos más adelante en el punto 2.4.

Una base de datos activa es capaz de ejecutar automáticamente acciones predefinidas en respuesta a eventos específicos siempre y cuando ciertas condiciones sean satisfechas.

La base de este esquema se encuentra en las "reglas activas", que presentan una estructura "evento – condición – acción". Las transiciones (o eventos, o cambios generado por un INSERT, UPDATE, ...) de la base de datos constituyen el corazón del modelo, pues son las que permiten detonar las reglas. La noción de transición constituye la base de Starburst. La transición es el "evento detonador".

Toda regla tiene:

- su identificador,
- una tabla, para la cual son relevantes las transiciones,
- un evento (o transición) frente al cual se activa,
- una condición de aplicación,
- una acción a tomar en el caso que la condición sea cierta, y
- un orden respecto a las otras reglas que conviven dentro de una misma base de datos.

Visto esto de una forma más rigurosa, pasamos a la sintáxis de las reglas (la notación utilizada seguirá siendo la de Starburst):

create rule *nombre* on *tabla*
when *evento*
[if *condición*]
then *acción*
[precedes *lista de reglas*]
[follows *lista de reglas*]

Mas allá de esta representación haremos explícitas algunas que por diferir con la arquitectura de implementación posterior serán discutidas más adelante.

La *condición* representa una consulta SQL de selección que de obtener alguna fila como resultado devuelve verdadero y ejecuta la *acción*. La *acción* es una sentencia SQL que se encuentre entre las siguientes: {insert, update, delete, rollback, select}. Las *listas de reglas*

indican la precedencia para el orden de ejecución. De no estar especificada la relación entre ambas, se tomará un algoritmo basado en la fecha de creación de las reglas para asignarles un orden de ejecución.

Ventajas y desventajas de la utilización de una base de datos activa.

Las bases de datos activas se están utilizando cada vez más en diferentes tipos de aplicaciones por diversos motivos:

- Posibilidad de introducir reglas de gestión del negocio centralizadas en un único lugar. Las reglas se almacenan en la base de datos y no se encuentran diseminadas en las diferentes componentes de la aplicación. Esto disminuye el costo de las modificaciones pues basta con adecuar el código residente en la base de datos sin necesidad de buscar otro lugar ni de recompilar otras componentes.
- Traslado de complejidad al motor de la base de datos, simplificando de esta forma la aplicación a desarrollar. La "factorización" producida al concentrar reglas de negocio (o una operación informática cualquiera) en un único lugar trae aparejada la simplificación mencionada. Ejemplo: todo registro de un evento de inserción en una tabla dada puede ser resuelto mediante una regla de la Base de Datos en lugar de tener que contemplar lo mismo en cada una de las transacciones que realice inserciones en dicha tabla.

Sin embargo diseñar un conjunto de reglas activas no es una tarea trivial, a este hecho hay que agregarle otras desventajas:

- Desconocimiento del mercado sobre bases de datos activas por tratarse de facilidades "modernas".
- Pocas herramientas disponibles para su cómodo manejo (edición, debug, documentación).
- Dificultades para el testeo, lo que implica que el testeo de una parte *crítica* de la aplicación es difícil de realizar (no resulta sencillo conocer el código ejecutado, las reglas disparadas, etc).

A través de esta tesis intentamos disminuir los puntos negativos enumerados anteriormente.

2.2 Testing

La pregunta que nos planteamos al iniciar una tarea de testing es la siguiente: Qué es lo que queremos testear? e inmediatamente después, cómo saber que hemos realizado nuestra tarea satisfactoriamente? Obviamente, el objetivo final es obtener algún tipo de "visto bueno" de los programas desarrollados. Si vemos a un programa como la representación de una función, uno de los objetivos del proceso de testing es asegurar que un programa realmente computa la función a la cual representa.

Para poder asegurar esta igualdad teórica (programa = función a computar), ciertos elementos son importantes y hay que tenerlos en cuenta [FP97]:

- El programa
- Una especificación del comportamiento esperado o un oráculo (Mecanismo de producción de respuestas correctas)

- Un método para observar el comportamiento del programa
- Una especificación de los "inputs" posibles
- Un método para determinar si el comportamiento observado coincide o no con el esperado.

En el marco de esta tesis nos limitaremos a trabajar para brindar el tercer punto citado anteriormente y facilitar la concreción del segundo.

2.2.1 Generalidades

Para responder a la pregunta presentada anteriormente en esta sección (¿corresponde el programa a la función a computar?), no existe un método práctico que nos asegure formalmente la correctitud y completitud de nuestro software, pero sí existen diferentes técnicas y estrategias de validación y verificación a aplicar entre las cuales se encuentra el testing.

Observación: El término *software* está cargado de ambigüedad, por la tanto dentro de este trabajo utilizaremos "software" (entre comillas) para representar a todo objeto efectivamente ejecutable (diferenciando el término de otro tipo de objetos que pueden ser simbólicamente ejecutados, a nivel del diseño por ejemplo)

Podemos dividir a las técnicas de validación y verificación en dos grandes grupos: las "estáticas" y las "dinámicas".

Las técnicas más básicas de validación y verificación de software son las denominadas "estáticas". Se denominan así pues analizan un software sin la ejecución del mismo. Dentro de las mismas están el chequeo de tipos y chequeo sintácticos, hoy cubiertos en su gran mayoría por los compiladores estándares del mercado. Todavía se utilizan los chequeos de escritorio (un programador leyendo el código fuente del programa en estudio; usualmente se realizan pruebas cruzadas, es decir un programador lee el código de otro programador), inspecciones grupales (guiadas por un moderador), y pruebas formales de correctitud (basadas en nociones de lógica).

Dentro de las técnicas estáticas de validación y verificación, tenemos también las siguientes categorías: análisis de flujo de datos, análisis de flujo de control y ejecuciones simbólicas, además de las técnicas generales enunciadas anteriormente.

Dentro de las técnicas denominadas "dinámicas" de validación y verificación (analizan un software a través de la ejecución del mismo), las técnicas de test son las más antiguas y tradicionales. También encontramos dentro de ésta categoría a las simulaciones, pruebas de stress, etc.

2.2.2 Testing

(Terminología básica obtenida de [ABC82]):

El proceso de testing consiste en obtener un valor posible de "input" (dato de entrada de un programa), determinar el comportamiento esperado, ejecutar el programa con el input seleccionado, observar el comportamiento real y por último, comparar los resultados. Si ambos resultados concuerdan, decimos que el test fue pasado satisfactoriamente; caso contrario, decimos que el test descubrió un error.

Observación: Para chequeos de robustez se pueden requerir "inputs" fuera del dominio de entrada válido.

Es importante acotar que cuando nos referimos a "test" incluímos a las cuatro categorías siguientes:

Test modular (o de unidad)
Testing que se limita al análisis individual de un módulo, viéndolo a éste como un todo.

Test de integración
Testing cuyo énfasis está puesto en la interacción entre los módulos y sus interfases.

Test de sistema
Testing cuyo énfasis está puesto en el sistema completo, viéndolo a éste como un todo.

Test de regresión
Testing cuyo énfasis está puesto en evitar los efectos colaterales de modificaciones introducidas en un sistema ya existente.

A pesar de la proliferación de las técnicas de testing, la única forma de asegurarse el objetivo buscado es mediante un test exhaustivo (ejecución del software con todos los inputs posibles). Obviamente, el dominio de inputs posibles puede y suele ser infinito o impracticable, por lo tanto el número de tests a realizar hace poco práctica la comprobación que un programa computa lo que deseamos por este camino. Se comenzó a investigar entonces diferentes propiedades y atributos de los desarrollos informáticos con el objetivo último de reducir los tests a realizar.

Generación de casos de Test

Como se puede vislumbrar una de las partes mas complicada del test reside en la elección del subconjunto de datos posibles de entrada al programa. Los datos elegidos deberán ser representantes de las diferentes "clases" de datos (llamadas "casos de test", conjunto de datos de entrada equivalentes para el proceso de Test.). Normalmente, el testeador sigue algún criterio, científico o intuitivo, para la elección de los datos con los cuales realizará su trabajo. Los resultados obtenidos para dichos casos son luego generalizados para el resto de los inputs posibles.

Dentro de los objetivos de máxima, se espera que la ejecución del programa para los casos de test, permita descubrir la máxima cantidad de errores existentes.

Descubrir un conjunto de datos apropiado a nuestros objetivos es una tarea delicada (ver [RW82]). En la práctica el conjunto de datos seleccionados deberán brindar al testeador una sensación de seguridad de que la mayor parte de los errores serán descubiertos lo que difiere del ideal que sería asegurar la correctitud del programa en estudio.

Podríamos imaginar un programa generador de casos de tests que dado el programa a testear como parámetro de entrada, devuelva un caso de test que respete el criterio predefinido. Se le puede solicitar también al programa imaginado que dado el caso de test encontrado, seleccione los "inputs" que lo satisfagan. Obviamente, la realización de este programa generador de tests es compleja, y en algunos casos, imposible (en el caso de programas no secuenciales el problema de seleccionar datos que satisfagan un caso de test no es decidible).

Es aquí también que se ve la importancia del producto desarrollado en el marco de esta tesis: el ADB_Tester. Con la ayuda de esta herramienta, se podrá guiar el testeador en la generación de sus datos de test.

Para elegir datos que respeten caminos (ver definición de "camino" en la página 15) elegidos se pueden usar, por ejemplo, técnicas de ejecución simbólica [Kin76].

Las técnicas de test se dividen a su vez en dos ramas: las "estructurales" y las "funcionales". El test estructural (noción de "caja blanca") es un método de test donde los casos de test son derivados de la estructura del código. El test funcional (noción de "caja negra") es un método de test donde los casos de test son derivados de la especificación funcional del software.

Si bien el ADB_Tester se limita exclusivamente al flujo de control (caja blanca), ya en [RW82] se afirma que el flujo de datos debe también ser tenido en cuenta. En [RW82], se desarrolla también un método para la obtención de casos más complejo.

Criterio de adecuación (o de cubrimiento)

Es una referencia que permite evaluar los casos de test efectuados. Corresponde a una propiedad que debe cumplir un conjunto de casos de prueba sobre el grafo de flujo de control.

Ejemplos:

- Cubrimiento de todos los caminos (toda instrucción debe ser ejecutada por lo menos una vez.
- Cubrimiento de todos los nodos
- Todo camino del flujo de control debe ser transitado por lo menos una vez.
- ...

Sin embargo, sea el criterio que sea el que uno elige, no tendrá asegurada la correctitud del programa [WO80].

Adecuación de casos de Test

Estudio que mide el grado de cubrimiento de un conjunto de casos de test sobre el software estudiado. Permite determinar cuando el soft ha sido probado lo suficiente.

El testing como actividad dentro del ciclo de vida del software está ligado a la noción de métrica [FP97] y en consecuencia es posible definir una métrica asociada. Dentro de los cubrimientos mas generales se encuentran los que intentan cubrir a todos los nodos existentes en el dbgraph, o a todos los ejes del dbgraph, o bien a todos los caminos (eventualmente con cierta propiedad) posibles del dbgraph en estudio.

Podemos definir entonces una métrica para evaluar el cumplimiento de un criterio. Por ejemplo, podemos definir el porcentaje de cumplimiento del criterio "cubrir a todos los nodos" como "(nodos transitados durante el test) dividido (nodos existentes)". Este tipo de métricas nos permiten comparar tests entre sí y evaluar al trabajo realizado.

Diferentes métricas pueden ser usadas para establecer estándares de calidad [FP97]. Se intenta catalogar, clasificar un producto con un nivel de calidad determinado, siempre y cuando supere en un porcentaje dado a la métrica preestablecida. Estas clasificaciones pueden resultar de gran utilidad en diversas situaciones, entre las cuales podemos mencionar "el nivel de seguridad requerido para pasar a producción un programa dado".

Terminación

Uno de los problemas a afrontar es saber cuando terminar el testing. No es posible conocer si los test realizados garantizan la ausencia de errores pero se debe tener algún parámetro para finalizar el trabajo y limitar así el costo del mismo.

No siempre es posible cumplir con un criterio de adecuación. De hecho el no cumplimiento de un criterio puede revelar errores. El ejemplo que ilustra la situación anterior es el de los caminos inalcanzables (o imposibles o no factibles) que son los que corresponden o contienen una secuencia de instrucciones que no serán nunca ejecutadas.

2.3 DBGraphs

Definiremos formalmente ciertas nociones que utilizaremos más adelante. Por cuestiones de comodidad con el trabajo realizado, nos hemos permitido redefinir ciertos conceptos teóricos.

- Definición de grafo:

Un grafo G es un par (N, E) dónde N y E son conjuntos, N no vacío y $E \subseteq N \times N$.

A los elementos del conjunto N, los denominaremos nodos y a los del E, ejes. En un elemento de E (n, m) denominamos a n "nodo origen del eje" y a m "nodo destino del eje".

- Grafo de una regla:

Se puede representar a cualquier programa secuencial por un grafo con las siguientes características. Los nodos representan una instrucción o un conjunto de instrucciones indefectiblemente secuenciales. Los ejes representan el flujo de control del programa. Esto es correcto pues dentro un mismo nodo no hay ningún tipo de bifurcación posible del control.

Presentamos un algoritmo que permite construir el grafo de una regla (con las diferencias al modelo teórico introducidas por Oracle) a partir del código fuente de la misma (en [Bal97] hay un algoritmo para el modelo de Starburst):

a. Toda regla tiene un único nodo inicial $n_{inicial}$

b. $n_{corriente} = n_{inicial}$

c. Tomo una palabra (toda secuencia de caracteres, diferentes al blanco, que se encuentra entre dos espacios en blanco) del código de la regla. Si la palabra es ...

   - *IF*, genero el siguiente subgrafo si la sentencia *IF* posee un *ELSE*:

   Grafo $SG = (NI, EI)$ con
   $NI = \{ n_{corriente}, n_{fin} \} \cup NTHEN \cup NELSE$ y
   $EI = \{(n_{corriente}, n_{then}), (n_{corriente}, n_{else}), (n_{finthen}, n_{fin}), (n_{finelse}, n_{fin})\} \cup ETHEN \cup EELSE.$

   NTHEN y *ETHEN* constituyen respectivamente los nodos y ejes del grafo *GTHEN* obtenido luego de aplicarle este mismo algoritmo al código que se encuentra entre el *THEN* y el *ELSE*. Denominamos n_{then} y $n_{finthen}$ a los nodos

inicial y final respectivamente del grafo *GTHEN* (podemos tener $n_{then} = n_{finthen}$).

NELSE y *EELSE* constituyen respectivamente los nodos y ejes del grafo *GELSE* obtenido luego de aplicarle este mismo algoritmo al código que se encuentra entre el *ELSE* y el *ENDIF*. Denominamos n_{else} y $n_{finelse}$ a los nodos inicial y final respectivamente del grafo *GELSE* (podemos tener $n_{else} = n_{finelse}$).

Si la sentencia *IF* no contiene *ELSE*

Grafo $SG = (NI, EI)$ con
$NI = \{ n_{corriente}, n_{fin}\} \cup NTHEN$ y
$EI = \{(n_{corriente}, n_{then}), (n_{finthen}, n_{fin}), (n_{corriente}, n_{fin})\} \cup ETHEN.$

NTHEN y *ETHEN* constituyen respectivamente los nodos y ejes del grafo *GTHEN* obtenido luego de aplicarle este mismo algoritmo al código que se encuentra entre el *THEN* y el *ENDIF*. Denominamos n_{then} y $n_{finthen}$ a los nodos inicial y final respectivamente del grafo *GTHEN* (podemos tener $n_{then} = n_{finthen}$).

Salteo (en todos los casos) la sentencia *IF* y asigno $n_{final} = n_{fin}$

- *LOOP*, genero el siguiente subgrafo:

Grafo $SG = (NL, EL)$ con
$NL = \{ n_{corriente}, n_{fin}\} \cup NLOOP$ y
$EL = \{(n_{corriente}, n_{loop}), (n_{finloop}, n_{fin}), (n_{corriente}, n_{fin}), (n_{finloop}, n_{loop})\} \cup ELOOP.$

NLOOP y *ELOOP* constituyen respectivamente los nodos y ejes del grafo *GLOOP* obtenido luego de aplicarle este mismo algoritmo al código que se encuentra entre el *LOOP* y el *ENDLOOP*. Denominamos n_{loop} y $n_{finloop}$ a los nodos inicial y final respectivamente del grafo *GLOOP* (podemos tener $n_{loop} = n_{finloop}$).

Salteo la sentencia *LOOP* y asigno $n_{final} = n_{fin}$

- Una diferente de todas las anteriores, $n_{final} = n_{corriente}$, y tomo la siguiente.

d. Si no llegué al final del código, vuelvo al punto c. con $n_{corriente} = n_{final}$.

e. No hay mas nodos ni ejes que los definidos anteriormente.

El grafo resultante modela a los flujos de control posibles de la regla en cuestión.

- ¿Qué es un DBGraph?

¡Un grafo muy particular!

Un DBGraph modela el flujo de control de las reglas de una base de datos activa y la relación entre las mismas. Está compuesto por la unión de los grafos de cada una de las reglas

de la base de datos, y el agregado de dos nodos; el A_i y el A_f. El primero de ellos (denominado nodo inicial), será relacionado con cada uno de los nodos iniciales de los grafos de las reglas que componen el DBGraph mediante un eje con origen en el nodo A_i; a estos ejes agregados los llamaremos ejes iniciales. El segundo, será relacionado con cada uno de los nodos finales de los grafos mediante un eje con destino en el nodo A_f; a estos ejes agregados los llamaremos ejes finales.

Por último, para completar la construcción del DBGraph, aplicaremos el siguiente algoritmo que puede generar ejes:

a) Pongo a todas las reglas de la base de datos en estado "NO_EXAMINADA"

b) Tomo una regla R_i cualquiera en estado "NO_EXAMINADA"

c) Recorro el código de la regla R_i hasta encontrar una de las siguientes palabras: "INSERT", "UPDATE", o "DELETE" no encontrada anteriormente.

d) Chequeo si la sentencia SQL encontrada puede detonar alguna regla R_j; si es el caso, genero un eje desde el nodo correspondiente a esa porción de código (ver algoritmo anterior) hacia el nodo inicial de la regla R_j.

e) Si no llegué al final del código de la regla R_i, vuelvo a c).

f) Si quedan reglas en estado "NO_EXAMINADA", vuelvo a b).

- Eje intra-regla: es aquel eje cuyo nodo origen y nodo destino corresponden a un grafo de una misma regla.

- Eje inter-regla o eje disparador: es aquel eje cuyo nodo origen y nodo destino corresponden a grafos de diferentes reglas. La generación de estos ejes se puede ver dentro del algoritmo presentado anteriormente.

Es importante recalcar que la existencia de un eje no implica que necesariamente se recorrerá esa porción de código asociado. Es decir que si un eje relaciona a un nodo representando a la porción de código anterior a un *IF* y a otro nodo representando al código correspondiente al *THEN* del mismo *IF*, el eje será transitado si la condición del *IF* es evaluada como TRUE.

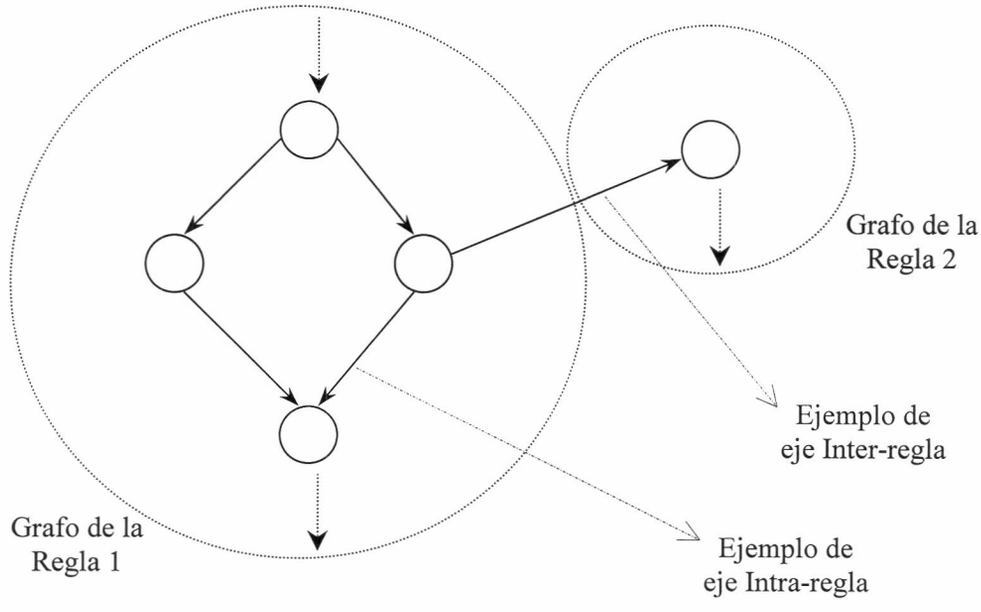
Otro caso un poco más complicado aparece en el caso de los ejes disparadores. Aquí, intervienen varios elementos para la determinación de tránsito de un eje: las condiciones de la regla, la eventual cláusula *WHERE* de la sentencia SQL, etc …

Ejemplo de definiciones vistas hasta este punto en la sección 2.2:

Regla 1	Regla 2
(Sobre la tabla TABLA_A)	(Sobre el INSERT de la tabla TABLA_B)

Código:
…
IF (…) THEN
…
ELSE
…
INSERT INTO TABLA_B …
…
END IF;
…
	Código:
	…

Gráficamente, los objetos definidos serían:



- Definición de camino:

Sea $G = (N, E)$ un DBGraph donde N es el conjunto de nodos y E es el conjunto de ejes. Un camino $P = e_0, e_1, \ldots, e_n$ es una secuencia de ejes donde el destino de e_i coincide con el origen de e_{i+1}. Cuando el destino de un eje A coincide con el origen de un eje B, decimos que A y B son ejes adyacentes.

- Definición de camino completo:

Sea $G = (N, E)$ un DBGraph donde N es el conjunto de nodos y E es el conjunto de ejes. Un camino completo $P = e_0, e_1, ..., e_n$ es un camino donde se cumplen las siguientes propiedades:

- el origen de e_0 es el nodo agregado A_i
- el destino de e_n es el nodo agregado A_f

2.4 Cubrimientos

Llegamos ahora a uno de los temas mas importantes con vistas al objetivo último a realizar: el testing de la base de datos activa. Habiendo estudiado los DBGraphs, pasamos ahora a definir un conjunto de propiedades sobre los mismos que nos permitirán evaluar el testing realizado.

- Definición de cubrimiento de nodos:

Sea $G = (N, E)$ un DBGraph y $C = \{P1, ..., Pn\}$ un conjunto de caminos. Decimos que C es un cubrimiento de nodos sobre G, si para todo nodo n_i que pertenezca a N, existe un eje e_j para algún Pk de C tal que el origen o el destino de e_j es n_i.

- Definición de cubrimiento de ejes:

Sea $G = (N, E)$ un DBGraph y $C = \{P1, ..., Pn\}$ un conjunto de caminos. Decimos que C es un cubrimiento de ejes sobre G, si para todo eje e_i que perteneza a E, existe un eje e_j en alguno de los Pi de C tal que $e_i = e_j$.

- Definición de cubrimiento de ejes disparadores:

Sea G = (N, E) un DBGraph y C = {P1, ..., Pn} un conjunto de caminos. Decimos que C es un cubrimiento de ejes disparadores sobre G, si para todo eje disparador e_i que perteneza a E, existe un eje e_j en alguno de los Pi de C tal que $e_i = e_j$.

- Definición de cubrimiento de ejes disparadores y ejes iniciales:

Sea G = (N, E) un DBGraph y C = {P1, ..., Pn} un conjunto de caminos. Decimos que C es un cubrimiento de ejes disparadores y ejes iniciales sobre G, si para todo eje inicial o disparador e_i que pertenezca a E, existe un eje e_j en alguno de los Pi de C tal que $e_i = e_j$.

Estas definiciones nos ayudarán a tener parámetros para la medición de la adecuación del testing realizado sobre la Base de Datos Activa a estudiar. Esto es así, pues un camino representa un comportamiento de la BDA frente a un conjunto de operaciones sobre la misma.

Siendo el objetivo final del testing expedirse sobre la bondad de la BDA, usaremos como medida de adecuación del testing realizado al porcentaje de cumplimiento de alguno de los cubrimientos definidos anteriormente.

Queda fuera del alcance de este trabajo el estudio de la base de datos a partir de la estructura del DBGraph resultante o la aplicación de algoritmos para la obtención de casos de test.

2.5 Oracle [Ora7a][Ora7b]

La industria incluyó rápidamente al concepto "base de datos activa". Decidimos crear una herramienta que funcione sobre Oracle. Las razones principales para tomar esta determinación fueron:

- Oracle constituye uno de los mas fuertes exponentes mundiales en bases de datos relacionales (se podría realizar un trabajo similar con Sybase, Progress, etc.),
- disponibilidad inmediata del software, y
- el conocimiento necesario para utilizarlo.

En primer lugar debemos estudiar como implementó Oracle la noción de base de datos activa.

Oracle simplificó el modelo de Starburst para poder brindarle al mercado una solución más práctica. Veamos entonces como son las reglas en Oracle, llamadas "database triggers" (o simplemente "triggers").

Los triggers no son más que procedimientos almacenados dentro de la base de datos e implícitamente ejecutados cuando una tabla es modificada (nunca explícitamente por un usuario o una aplicación, aquí la principal diferencia con otro tipo de procedimientos almacenados dentro de la base de datos). Las modificaciones relevantes para la detonación de los triggers son: INSERT, UPDATE o DELETE (inserción, modificación o borrado) sobre alguna tabla, y no sobre una vista ("view"). Esta distinción entre tabla y vista es importante pero hay que saber que una acción sobre una vista puede detonar colateralmente un trigger sobre la tabla que sirve de base para la vista.

El código que pueden almacenar los triggers es SQL o bien PL/SQL y pueden referenciar explícitamente a procedimientos o funciones almacenados dentro de la base de datos.

Los usos más frecuentes asignados a los triggers (¡ésto cobra relevancia para determinar dónde puede ser mas útil nuestra herramienta!) de acuerdo Oracle son los siguientes:

- Generación automática de los valores de columnas derivadas.
- Prevención de transacciones inválidas durante ciertas horas.
- Reforzar sistemas de autorización.
- Reforzar controles de integridad en bases de datos distribuidas.
- Complementar reglas de negocio complicadas.
- Registrar eventos en forma transparente para la aplicación.
- Mantener replicaciones sincrónicas de tablas.
- ...

En lo que resta de la sección 2.5 estudiaremos a los triggers en profundidad. Comenzaremos en la sección 2.5.1 estudiando las componentes que tienen los triggers en Oracle. Veremos también en la sección 2.5.2 al modelo de ejecución y en la 2.5.3, un conjunto de restricciones llamativas. Por último, dedicamos 2 secciones (2.5.4 y 2.5.5) a detallar cuestiones importantes al momento de la implementación del ADB_Tester: ¿cómo almacena Oracle a sus triggers?, ¿cómo es el código dentro de un trigger?.

2.5.1 Estructura de los triggers

Los triggers están compuestos por tres partes principales: el evento detonador, una restricción, y una acción. Veamos que tiene cada una de las componentes de un trigger y cuáles son sus relaciones con el modelo teórico.

Evento detonador

Se trata de la sentencia SQL que detona el trigger. Obviamente puede tratarse también de la unión de varias sentencias SQL (sentencia_A, sentencia_B, ..., sentencia_N) y en consecuencia el trigger se detonará frente a la sentencia sentencia_A, a la sentencia_B, ..., y a la sentencia_N. Las sentencias deben hacer referencia a una única tabla.

Restricción

Se trata de una expresión lógica que, sólo de evaluarse en 'Verdadero', produce la detonación del trigger. El objetivo de esta funcionalidad es el de condicionar la ejecución de un trigger. Esta posibilidad se encuentra sólo disponible para los triggers que se detonan para cada fila (ver más adelante los diferentes tipos de trigger que hay).

La restricción en cuestión reemplaza a la condición de aplicación vista en la sección 2.1.

No utilizaremos dentro de nuestra herramienta de parseo del código de los triggers a esta facilidad, pues la estructura del grafo que modela al trigger no se ve modificado por esta componente. Esto se debe a que la restricción se evalúa en tiempo de ejecución y hemos decidido realizar un estudio estático del código.

Acción

Es el procedimiento que contiene las sentencias SQL o el código PL/SQL que se ejecutará frente a un evento detonador siempre y cuando la expresión de restricción sea 'Verdadera'.

Como vemos, dichas posibilidades difieren del modelo original propuesto que sólo aceptaba una sentencia SQL. La otra diferencia radica en la imposibilidad de realizar un rollback explícito de las acciones llevadas a cabo, así como de llevar a cabo un commit.

Tipo de trigger

Al definir un trigger podemos especificar el número de veces que se ejecutará frente a algún evento. Tenemos dos posibilidades: una por cada fila afectada por el evento detonador (denominado 'row trigger'; por ejemplo una vez por cada fila modificada por una sentencia de UPDATE) o bien una única vez por evento detonador sin importar la cantidad de filas involucradas (denominado 'statement trigger').

No utilizaremos dentro de nuestra herramienta esta facilidad, pues carece de relevancia para el tipo de test que realizaremos. La estructura del grafo que modela al trigger no se ve modificado por esta componente pues el tipo de trigger se utiliza para determinar el tiempo de ejecución y hemos decidido realizar un estudio estático del código.

Cláusulas Before y After

Al definir un trigger podemos también determinar el momento de ejecución. Tenemos aquí también dos posibilidades: antes (BEFORE) de llevar a cabo el evento detonador o bien luego (AFTER) de llevarlo a cabo.

No utilizaremos dentro de nuestra herramienta esta facilidad, pues carece de relevancia para el tipo de test que realizaremos. La estructura del grafo que modela al trigger no se ve modificado por esta componente.

Estas componentes se relacionan de la siguiente forma:

```
CREATE [OR REPLACE] TRIGGER [schema.]trigger
  {BEFORE | AFTER}
  {DELETE | INSERT | UPDATE [OF column [, column] ...]}
  [OR {DELETE | INSERT | UPDATE [OF column [, column] ...]}] ...
  ON [schema.]table
  [ [REFERENCING { OLD [AS] old [NEW [AS] new]
            | NEW [AS] new [OLD [AS] old] } ]
   FOR EACH ROW
   [WHEN (condition)] ]
  pl/sql_block
```

Se observa la desaparición de la posibilidad teórica de asignarles una relación de orden a las reglas dentro de Oracle. Veremos en la sección siguiente cómo se trabaja a la noción de "orden".

2.5.2 Ejecución de los triggers

Un trigger puede hallarse en dos estados diferentes: 'habilitado' ('ENABLED') o 'deshabilitado' ('DISABLED'). La información de relevancia en el marco de esta tesis se limita a saber que Oracle se encarga de la ejecución de los triggers y no posee un orden de ejecución asignado por el usuario para diferentes triggers relacionados a mismos eventos externos.

Una base de datos relacional no garantiza el orden en que los diferentes registros son procesados por una sentencia SQL. En consecuencia, no se deben crear triggers que dependan del orden de procesamiento de los registros. Para el orden de ejecución de los diferentes triggers, el orden de ejecución es uno arbitrario determinado por Oracle (usualmente basado en la fecha de creación de los mismos). En consecuencia para asegurarnos la ejecución ordenada de las reglas de la base de datos se recomienda juntar al código de los diferentes triggers detonables por el mismo evento, dentro de un único trigger. Así se le puede asignar el orden deseado al código.

La única secuencia garantizada por Oracle para la ejecución de triggers asociados al mismo evento SQL es la dada por el siguiente modelo:

1. Ejecución del 'before statement trigger'
2. Mientras haya filas afectadas por la sentencia SQL detonadora
   a. Ejecución del 'before row trigger'.
   b. Modificación del registro y chequeo de las restricciones de integridad.
   c. Ejecución del 'after row trigger'.
3. Finalización del chequeo de las restricciones de integridad.
4. Ejecución del 'after statement trigger'.

El modelo anterior es recursivo, ésto es importante saberlo pues recordemos que un trigger puede llegar a detonar a otro trigger.

Una propiedad importante del modelo presentado es que todas las acciones y todos los chequeos realizados a consecuencia de una sentencia SQL deben ser exitosos. Caso contrario, se deshacen todas las modificaciones realizadas a la base de datos inducidas por la sentencia SQL original.

Oracle no provee demasiadas facilidades para la realización de diagnósticos durante la ejecución de los diferentes triggers. La forma estándar que provee Oracle para el Debug es la utilización del procedimiento DBMS_OUTPUT.PUT o PUT_LINE con los parámetros que se desean mostrar por pantalla. Una forma un poco mas elaborada de seguir un programa es mediante la utilización del Oracle Procedure Builder (que forma parte del Oracle Developer 2000). Este producto brinda ciertas facilidades para la ejecución de triggers dentro de un ambiente controlado en el cual se pueden colocar "breakpoints", visualizar el contenido de las variables, etc.

2.5.3 Restricciones en los triggers

El código del bloque de PL/SQL que tolera un trigger tiene ciertas limitaciones con relación al PL/SQL estándar. Veamos cuáles son las diferencias.

Sentencias SQL válidas en el cuerpo de un trigger

Se pueden utilizar sentencias de manipulación de datos pero no las de definición. Tampoco se permiten las sentencias de control de transacción como ser ROLLBACK, COMMIT, y SAVEPOINT.

Restricciones sobre los tipos LONG y LONG RAW

- Una sentencia SQL de un trigger puede insertar datos en una columna de una tabla del tipo LONG o LONG RAW. Lo que no se permite es la recuperación de dicha columna. La sóla posibilidad para su utilización es la conversión de la columna a un tipo soportado (ejemplo VARCHAR2(32000)).
- No se pueden declara variables del tipo LONG o LONG RAW.
- No se pueden utilizar los operadores :NEW y :OLD en combinación a columnas de tipo LONG o LONG RAW.

Oracle no conservó:

- Eventos temporales que detonen reglas.

Propiedades garantizadas por Oracle:

- *Terminación:* la detonación de triggers producida por otros triggers puede no tener fin; cuando se sobrepasan ciertos límites el motor de la base de datos devuelve un código de error. Esto nos asegura una terminación de nuestro proceso, sin embargo, dependiendo del límite mencionado puede resultar en una limitación. La explicación de esta propiedad, se encuentra en la implementación dada para los triggers. Para la misma se utilizan cursores, y la cantidad máxima de cursores abiertos por sesión está limitada (usualmente a 32). El límite es parametrizable.

- *Confluencia:* asegurada por las restricciones dadas en el párrafo siguiente sobre las "mutating tables" y las "constraining tables" por un lado y también por la imposibilidad de tener dos reglas asociadas a un mismo evento.

Una "mutating table" es una tabla que está siendo modificada por una sentencia de UPDATE, DELETE o INSERT en este preciso momento, o bien una tabla que puede necesitarse su modificación por los efectos de un borrado en cascada declarado como restricción de integridad en la base de datos (DELETE CASCADE). Una tabla restringida (constraining table) es una tabla cuyo contenido puede necesitarse desde una sentencia SQL en forma explícita o bien implícitamente por una restricción de integridad. Las nociones introducidas están limitadas a la sesión de trabajo perteneciente a la sentencia en curso.

Sobre este tipo de tablas existen dos restricciones puestas por Oracle para asegurar la coherencia de la base de datos:

1. Las sentencias SQL de un trigger no puede leer o modificar una "mutating table" de la sentencia detonadora del trigger.
2. Las sentencias de un trigger no pueden modificar las columnas definidas como PRIMARY, FOREIGN o UNIQUE KEY de una tabla restringida de la sentencia detonadora del trigger.

Existe una excepción a las restricciones anteriores. Los triggers del tipo BEFORE ROW y AFTER ROW detonados por una sentencia de INSERT de un único registro sobre una tabla, no tratan a ésta como "mutating" o restringida.

Si se intenta vulnerar a dicha situación, Oracle devuelve un mensaje de error durante la ejecución y los efectos del trigger así como los de la sentencia detonadora son anulados, devolviéndole el control al usuario o a la aplicación.

2.5.4 Almacenamiento de los triggers de Oracle

Para el aspecto implementativo del proyecto en último lugar debimos estudiar al diccionario de datos de Oracle para estar en condiciones de hacer la herramienta buscada. Dentro del diccionario de datos, la información esencial es el almacenamiento de los triggers: la tabla USER_TRIGGERS. Dicha tabla contiene a la información más relevante de las reglas.

Obviamente todo trigger posee un identificador "TRIGGER_NAME" (de hasta treinta caracteres, pero notar que para nosotros deben ser menos veinte). También posee la siguiente información:

- Un atributo "TRIGGER_TYPE" que indica si el trigger se ejecuta antes o después de cada registro involucrado en la operación; es el tipo de trigger.
- Un atributo "TRIGGERING_EVENT" que indica frente a que acción sobre la entidad en cuestión se ejecutará la regla.
- Propietario de la regla (Usuario Oracle indicado en "TABLE_OWNER")
- Nombre de la tabla sobre la que está basada la regla ("TABLE_NAME")
- Denominación para los valores nuevos y antiguos dentro de las reglas. Los valores por default son "old" y "new" y no permitimos variaciones al respecto.
- La restricción de la regla está representada por el atributo "WHEN_CLAUSE"
- Un estado que permite habilitar o deshabilitar a una regla dada (enabled/disabled). Dicho estado se representa con el atributo "STATUS".

- Una descripción de la regla en cuestión.
- El código del trigger propiamente dicho que se almacena en el campo "TRIGGER_BODY".

La otra tabla del diccionario de datos utilizada para la construcción de la herramienta es la USER_TRIGGER_COLS, donde se almacena la información referida a la modificación de las tablas (el "update"). A través de esta tabla, se obtienen los campos que siendo modificados en una tabla que posee reglas asociadas, detona a los mismos. Sus campos mas relevantes son los siguientes:

- "TRIGGER_OWNER"
- "TRIGGER_NAME"
- "TABLE_OWNER"
- "TABLE_NAME"
- "COLUMN_NAME"
- "COLUMN_LIST"

Oracle provee funciones para la administración de dichas tablas recomendando fuertemente no modificar directamente el diccionario de datos. Además, si bien Oracle almacena sólo el código del trigger, luego de la primera ejecución utiliza el compilado que almacena en memoria, y en consecuencia, si modificamos el diccionario de datos sin pasar por la herramienta adecuada, no observaremos cambio alguno hasta tanto Oracle haya decidido cambiar el contenido de su memoria.

Para la creación, modificación o eliminación de los triggers se cuenta con instrucciones de SQL para llevarlas a cabo.

2.5.5 Sintáxis del PL/SQL de Oracle

Es importante para la etapa del desarrollo de la herramienta, el estudio del PL/SQL utilizado para la escritura del código de las reglas. No resulta de interés dentro del marco de este trabajo conocer a fondo la sintáxis del código de las reglas, pero sí es necesario un conocimiento de la misma para estar en condiciones de realizar el desarrollo de la herramienta.

Sintáxis

```
plsql_block ::=

        [DECLARE
          object_declaration [object_declaration] ...
          [subprogram_declaration [subprogram_declaration] ...]]
        BEGIN
          seq_of_statements
        [EXCEPTION
          exception_handler [exception_handler] ...]
        END [label_name];


object_declaration ::=
```

{ constant_declaration
| cursor_declaration
| cursor_variable_declaration
| exception_declaration
| plsql_table_declaration
| record_declaration
| variable_declaration}

subprogram_declaration ::=

{function_declaration | procedure_declaration}

Descripción de las palabras clave mas importantes:

DECLARE

Esta palabra clave señala el comienzo de la sección de las declaraciones locales de un bloque de PL/SQL. Esta sección es optativa y termina implícitamente con la palabra clave BEGIN que introduce a su vez a la sección ejecutable del bloque.

BEGIN

Señala el comienzo del código PL/SQL ejecutable, de las sentencias que serán efectivamente ejecutadas. Esta porción de código es obligatoria (la sentencia NULL puede cumplir con dicho requisito).

seq_of_statements

Representa a una secuencia de sentencias ejecutables (por contraposición a las sentencias declarativas), entre las cuales se pueden encontrar sentencias SQL y bloques de PL/SQL. La sintáxis del seq_of_statements es la siguiente:

sentencia [sentencia] ...

Las sentencias son usadas para la creación de algoritmos. Además de las sentencias SQL, el PL/SQL posee sentencias de control de flujo y sentencias para el manejo de los errores. El formato de las sentencias en PL/SQL es libre, ésto significa que puede haber una por línea, todas seguidas, etc. La sóla restricción está dada por el punto y coma (";") que imperiosamente las concluye. La sintáxis de las sentencias es la siguiente:

sentencia ::=

{ assignment_statement
| exit_statement
| goto_statement
| if_statement
| loop_statement
| null_statement
| plsql_block
| raise_statement

| return_statement
| sql_statement}

PL/SQL soporta a un subconjunto de las sentencias SQL que incluye a la manipulación de los datos, manejo de cursores, control de la noción de transacción pero excluye a la definición de datos y sentencias de control de datos como ser ALTER, CREATE, GRANT, and REVOKE. La sintáxis del sql_statement es la siguiente:

sql_statement ::=

{ close_statement
| delete_statement
| fetch_statement
| insert_statement
| lock_table_statement
| open_statement
| open-for_statement
| select_statement
| update_statement}

EXCEPTION

Esta palabra clave marca el comienzo del bloque PL/SQL encargado de procesar errores de excepción. Cuando una excepción ocurre, la ejecución normal del bloque se ve interrumpida y el control se transfiere inmediatamente al controlador de excepciones correspondiente. Una vez finalizada la ejecución del nuevo bloque, la ejecución continua en la instrucción siguiente a la de la interrupción. Este tipo de situaciones quedan fuera del alcance de nuestro trabajo.

END

Esta palabra clave marca el fin del bloque PL/SQL. Debe ser la última palabra del bloque. Ni el END IF de una sentencia IF o el END LOOP una sentencia LOOP pueden sustituir la palabra clave END.

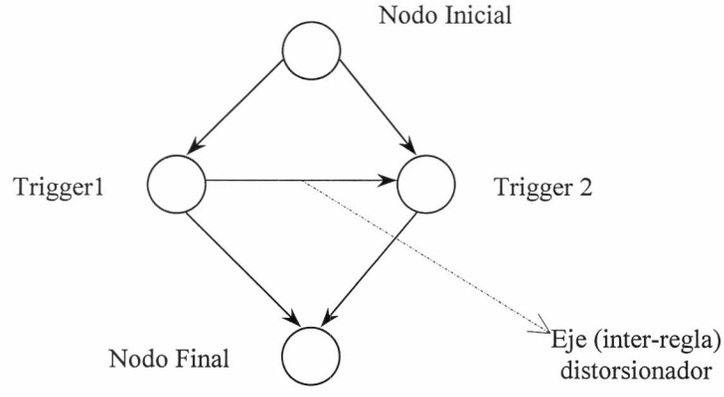
Esto fue la sintáxis teórica de los bloques de PL/SQL; sin embargo, hemos decidido simplificar la sintáxis de nuestros triggers eliminando a las "exceptions" como parte del código PL/SQL disponible para los triggers. En el caso de haber triggers con "exceptions" podríamos tener alterados los caminos existentes dentro del grafo que los modeliza. Dentro de los efectos negativos que ocasionaría un trigger con una "exception" tenemos el caso de sentencias de INSERT, UPDATE o DELETE que detonen a otros triggers. En esta situación, aparecería un camino rara vez transitado en la realidad pero visible en forma permanente a través del grafo obtenido con nuestra herramienta!

La "exception" es un quiebre atípico en el flujo de control similar al producido por un "GOTO".

Veamos el ejemplo que ilustra este problema:

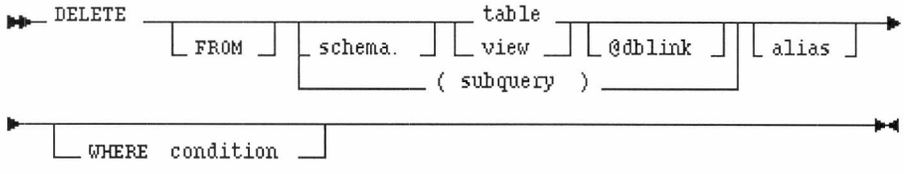
Trigger 1	Trigger 2
(Sobre la tabla TABLA_A)	(Sobre el INSERT de la tabla SOFTLOG)
Código: … select sueldo from empl …; … exception when no_data_found then INSERT INTO SOFTLOG …; End;	

El DBGraph asociado a una base de datos con estos dos triggers sería el siguiente:



SENTENCIAS SQL DE INTERES PARA NUESTRO PARSER

Sintáxis del DELETE



Parámetros involucrados

schema

se trata del esquema que contiene a la tabla o vista (atención no se pueden utilizar triggers directamente sobre una vista). Si no se encuentra especificado, Oracle7 asume que la tabla está en nuestro propio esquema.

table / view

es el nombre de la tabla o vista de la cual se desea eliminar registros.

dblink

es el nombre del lazo de la base de datos con una remota que contiene a la tabla o vista. En caso de omisión, Oracle7 asume que la tabla o vista se encuentra en la base de datos local.

subquery

se trata de una subconsulta que permite extraer el conjunto de datos a eliminar.

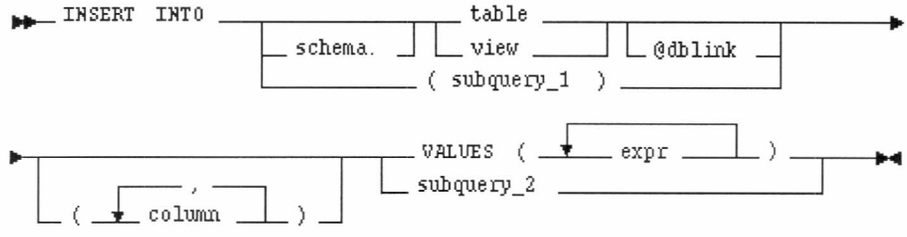
alias

se trata de un alias asignado a la tabla, vista o subconsulta.

WHERE

Condición que deben cumplir las filas para ser eliminadas.

Sintáxis del INSERT



schema

se trata del esquema que contiene a la tabla o vista (atención no se pueden utilizar triggers directamente sobre una vista). Si no se encuentra especificado, Oracle7 asume que la tabla está en nuestro propio esquema.

table view

es el nombre de la tabla o vista sobre la cual se desea insertar registros.

dblink

es el nombre del lazo de la base de datos con una remota que contiene a la tabla o vista. En caso de omisión, Oracle7 asume que la tabla o vista se encuentra en la base de datos local.

subquery_1

se trata de una subconsulta que será utilizada como una vista. No será utilizada en nuestro trabajo.

column

se trata de la columna de la tabla o vista.

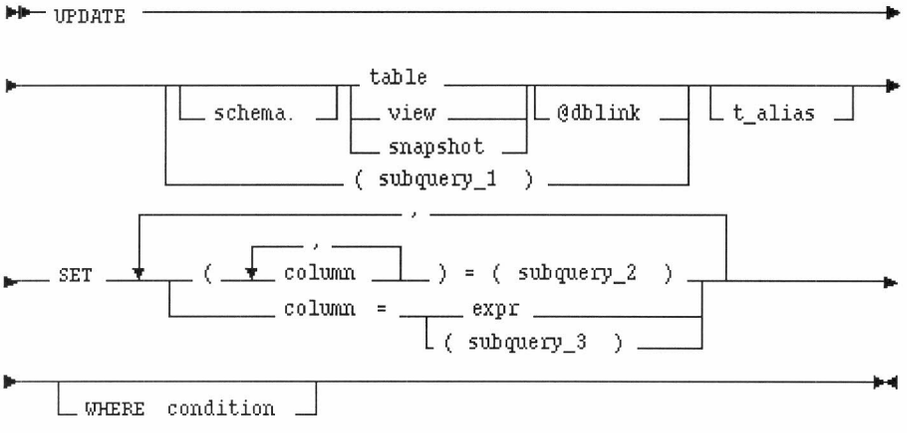
VALUES

Especifica los valores de la fila a insertar

subquery_2

es una subconsulta que devuelve las filas a insertar en la tabla.

Sintáxis del UPDATE



3 Metodología

Una buena metodología de testeo debe cubrir todo el ciclo de vida del software. Sin embargo en este trabajo no estamos interesados en abarcar el ciclo de vida entero, sino que nos limitamos a la verificación de ejecución del código implementado dejando de lado etapas importantes que cubren entre otras el análisis, el diseño o bien cuestiones propias de la implementación como ser el respeto por las normas de programación propias de un grupo de trabajo.

El interés de nuestro trabajo se pone de relieve cuando las reglas tienen mucha interacción entre sí. Para casos de bases de datos "no muy activas" (es decir sin demasiadas reglas) o bien para bases de datos activas "no muy complejas" (es decir sin demasiadas relaciones entre las diferentes reglas o sin sentencias de un trigger que detonen eventualmente a otros triggers) la metodología presentada puede presentar demasiada carga de trabajo. Esta carga de trabajo generaría un "over head" indeseable frente a una ganancia quizás exigua en este contexto: entender una aplicación ya simple y determinar las reglas ejecutadas (muy probablemente una sola).

Veremos en este capítulo una clasificación de la metodología desarrollada, cuál es el esquema de testeo propuesto para la BDA, cuáles son los factores a tener en cuenta para la realización del testeo y finalmente, cómo ejecutar realmente el esquema de testeo propuesto.

3.1 Clasificación

Según hemos descripto anteriormente en este trabajo, existen diferentes clasificaciones para los métodos de validación y verificación del software. La metodología propuesta es básicamente una metodología "dinámica" al tratarse de "Testing de Bases de datos Activas".

La metodología desarrollada se basa en el principio de caja blanca (descrito en la sección 2.3.1). Este enfoque, como vimos en el capítulo anterior involucra al testeador con el software subyacente y su estructura. La construcción del DBGraph así lo atestigua.

Nuestra metodología de test constituye un paso más en el análisis del flujo de control del software. Apuntamos a facilitar el análisis de cubrimientos.

Aplicación de la metodología para las cuatro categorías de Test vistas

Con respecto del alcance del testing realizado, no podemos catalogarla como una herramienta de *test de sistema*, ya que en el test no solo intervienen todas las reglas que componen la aplicación. Es habitual encontrar que el diseño de las bases de datos activas está totalmente dispersado en su concepción y las relaciones entre diferentes reglas provocan efectos colaterales (a veces indeseados); la utilización de la metodología en los tests de sistemas puede compensar esta falencia.

Por otro lado, se la podría considerar también como una herramienta apta para el *test modular*, pues se limita sólo a la parte de la base de datos activa, sin tener una visión más amplia del conjunto de la aplicación. Este es su ámbito natural.

Por último, además podemos utilizar a la metodología desarrollada para la realización de los *tests de regresión*. En estos tests, para el ámbito de las bases de datos activas, es conveniente cerciorarse que las modificaciones introducidas en las reglas no detonan reglas que no deberían.

Antes de desarrollar las etapas de nuestra metodología, conviene reiterar que el testing no asegura la ausencia de errores de un programa sino que brinda un gran filtro para su disminución. Gracias a la metodología desarrollada se aumenta la confiabilidad del programa sometido al testing.

3.2 Esquema de testeo de reglas

3.2.1 Planificación

La planificación del testing se apoya en el conocimiento de diferentes técnicas de testeo desarrolladas en el Capítulo 2 en un intento de sobreponerse a las prácticas habituales que se basan en enfoques personales fuera de bases científicas sólidas.

Cuando se empieza a testear una aplicación, el testeador se puede enfrentar con dos situaciones: se posee conocimiento funcional y técnico de la aplicación a testear o no. Para el último de los casos estaremos brindando un excelente punto de partida. Para el experto que ya conoce la aplicación a testear puede ser que esta herramienta lo ayude a estructurar y ordenar sus pruebas y a descubrir ciertas falencias (ejemplo: caminos inalcanzables o partes no testeadas por él)

Veremos más adelante (ver caso de estudio teórico en la sección 5.2) que el conocimiento funcional puede ser de vital importancia para la realización de un test significativo que traspasa las fronteras del aspecto estructural de un código fuente.

3.2.1.1 Elementos del Testing

Para una efectiva planificación de la etapa de Test se requiere de la presencia de diferentes elementos, a saber:

o) *La Base de Datos Activa*, para poder realizar efectivamente los Tests y no quedar limitados a un trabajo teórico sin corroboraciones empíricas.

i) *Visualización del grafo entregado por el ADB_Map*, para poder elegir los caminos a recorrer, las métricas a usar, los casos de Test existentes, comparar diferentes alternativas, etc (ver casos de estudio en el capítulo 5)

ii) *Diseño Lógico de la BDA*, para poder instanciar las sentencias SQL que permitirán alcanzar el cubrimiento propuesto.

iii) *Variables de entorno* que inciden sobre el Testing a realizar (tiempo disponible, recursos disponibles, ...), para tomar la decisión sobre le tipo de Test a realizar, su profundidad y confiabilidad.

Con estos elementos se deberá elegir la métrica a utilizar para la evaluación del test, el cubrimiento deseado y el criterio de éxito que le aplicaremos a nuestro trabajo. Tenemos en definitiva las armas que nos permitirán elegir la mejor estrategia de testing.

3.2.1.2 Determinación de los casos y datos de Test

Para lanzarse definitivamente en la ejecución práctica del Test, se deben realizar con anterioridad ciertas tareas. Se deben determinar el conjunto de casos y datos de test y los procesos que permitirán alcanzar el cubrimiento esperado. La obtención de los datos es un tema que no se desarrollará en el marco de esta tesis, sin embargo existen diferentes posibilidades. Una de ellas es que el testeador seleccione casos y datos de acuerdo al uso esperado de la base de datos.

3.2.2 Ejecución

¿Cuál es el esquema que se propone en este trabajo para la realización del Testing de Bases de Datos Activas?

En primer lugar se debe conocer la estructura "activa" de la base de datos a testear, ya que es en base a esta, que podemos obtener un plan estratégico de testing. Esto equivale a obtener el DBGraph asociado a la base de datos. El DBGraph de nuestra base de datos activa obtenido, es el requerido en la sección 3.2.1.1 de Planificación.

En segundo lugar se debe ejecutar la aplicación con los diferentes juegos de prueba seleccionados. Pero ejecutar la aplicación no es suficiente, también es necesario obtener un registro de las pruebas efectivamente realizadas (que no concuerdan necesariamente con las pruebas planificadas pues dos caminos diferentes pueden bastar para cumplimentar un cubrimiento dado). Para esto se debe preparar la base de datos poniéndola en modo "debug".

En tercer lugar, cuando haya finalizado la ejecución de las pruebas estaremos en condiciones de estudiar su resultado. Nos encontramos ahora con el problema de recuperar la información relevante sobre el test realizado, pues si no apareció ningún error a la vista del probador no podemos asegurarnos de la bondad del testing. De no mediar algún paso más en la metodología diseñada, no se podría medir el testing realizado; sólo habríamos recibido una ayuda para la planificación del testing. Poseemos una manera de visualizar el cubrimiento obtenido a partir de los tests realizados, esto es una herramienta para obtener el DBGraph con los ejes transitados marcados en color.

Una vez analizada la representación gráfica del DBGraph ejecutado se puede decidir si es necesario ejecutar nuevos tests para aumentar el cubrimiento obtenido. Si es así, no necesitamos volver a comenzar todo el proceso desde el inicio, es suficiente con definir los tests deseados, y volver al paso anterior. Repetiremos este procedimiento hasta que el cubrimiento sea satisfactorio.

Por último, hay que poner a la base de datos en modo "normal" (sacarla de modo "debug").

Para efectuar el análisis del testing realizado, se presenta como documentación una copia de las reglas existentes en la base de datos, la especificación del juego de datos, las pruebas realizadas y el grafo resultante (estructura de la aplicación con los caminos recorridos)

Es importante no olvidar realizar el proceso completo sin omitir ningún paso indicado en la metodología, para posibilitar la correcta interpretación de los resultados y que el estado de la base de datos no se vea alterado por el testing.

Para realizar cualquier modificación en las reglas de la base de datos (los triggers), esta debe estar en modo normal. Una vez introducidos los cambios en las reglas, se deberá comenzar el proceso de testing desde el primer paso. La causa de ésto es simple: al introducir cambios en la codificación de las reglas (para eliminar errores de codificación o bien a raíz de una modificación de las especificaciones) posiblemente se introduzcan cambios en el DBGraph resultante, pues los cambios se pueden hallar en dos niveles diferentes:

- En la estructura del grafo que representa la aplicación, es decir que puede haber nodos y/o ejes agregados o eliminados.
- En la secuencia de ejecución del DBGraph que genera un conjunto de caminos recorridos diferente (se visualiza como una coloración del grafo diferente).

3.2.3 Análisis

Se deben comparar ahora los resultados previstos con los obtenidos. En caso de diferencia ... eureka, podremos mejorar nuestro aplicativo buscando y corrigiendo las causas de la diferencia encontrada; y en el caso de obtener resultados coincidentes ... eureka, seremos conscientes del testeo realizado y del no realizado (es diferente decidir no testear una parte de la aplicación que obviar su testeo).

3.3 Ejecución de la metodología

Veremos en esta sección como realizar *concretamente* los diferentes pasos descriptos en la sección anterior 3.2.

3.3.1 Planificación del testing

Para ejecutar el testing es necesario codificar los experimentos diseñados. Esta fase puede entenderse como la escritura de verbos SQL, la utilización de ciertas transacciones que detonarán los verbos SQL o mas genéricamente como cualquier método que sirva para detonar eventos en la base de datos.

Si es necesario se generarán scripts que provean los datos necesarios para ejecutar el testing. Por ejemplo: si los triggers toman en cuenta los valores de algunas tablas, debe verificarse el contenido de las mismas.

3.3.2 Ejecución real de los casos de Test

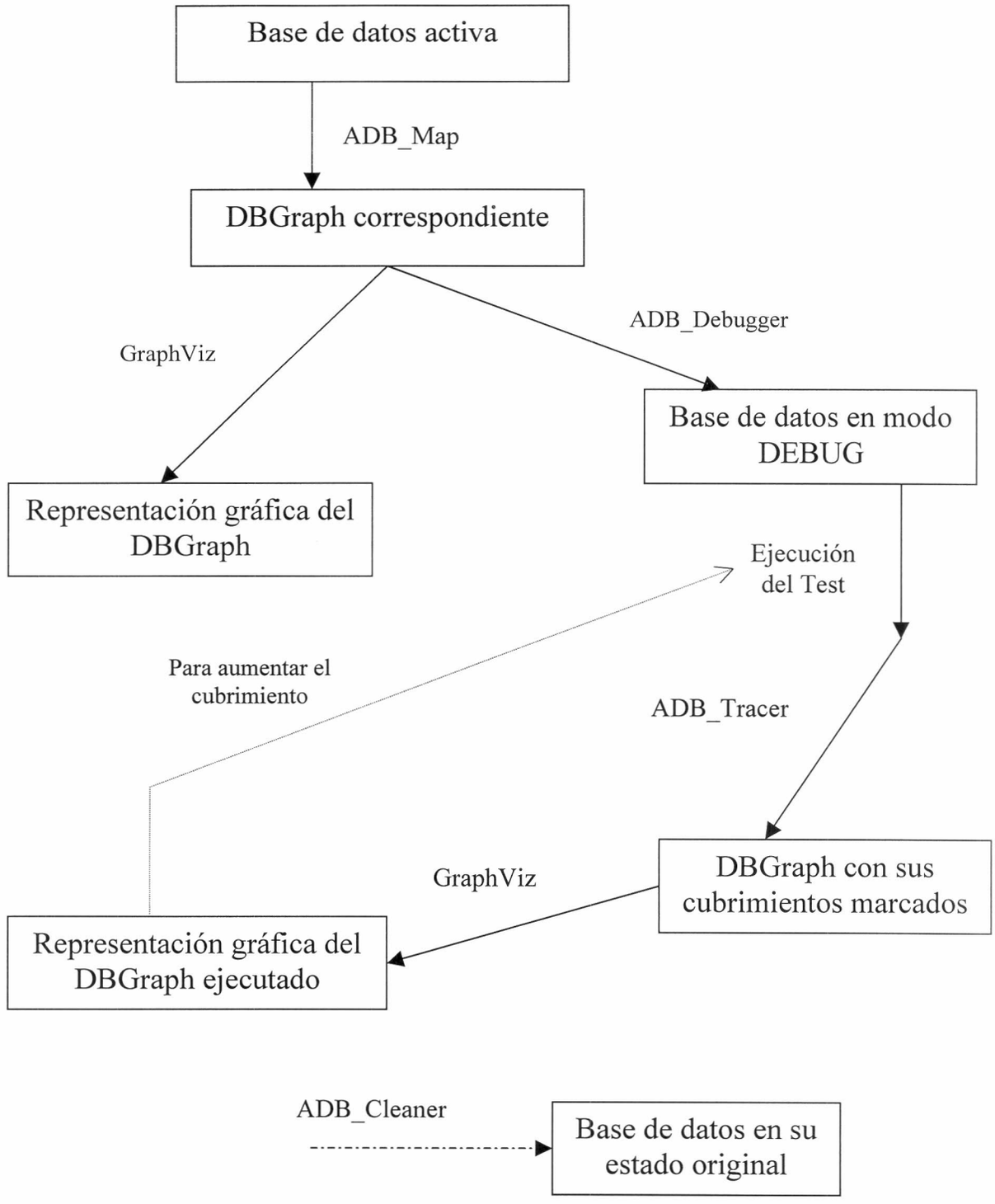
Para facilitar la interpretación de los grafos se utiliza una herramienta de graficación para MS-Windows, GraphViz. Esta recibe como entrada un archivo ASCII con la especificación del grafo. Todos los utilitarios desarrollados en el marco de esta tesis que obtienen algún tipo de grafo, generan un archivo con la especificación según lo requiere GraphViz.

El método para obtener el DBGraph buscado es ejecutar el utilitario ADB_Map, que genera un archivo ASCII con las especificaciones del mismo.

El método para poner a la base de datos en modo "debug" es ejecutar el utilitario ADB_Debugger, que modifica los triggers de la base para que generen un rastro de las partes de la aplicación efectivamente recorridas.

Para obtener la representación visual de los resultados, es necesario ejecutar el utilitario ADB_Tracer, que modifica el archivo generado por ADB_Map, marcando los caminos efectivamente transitados. Cuando se visualice el grafo, dichos ejes aparecerán en color rojo.

Ejecutando el utilitario ADB_Clean se restablece el modo "normal" de la base de datos.



El presente cuadro sintetiza el esquema establecido para realizar el testing de bases de datos activas

3.3.3 Análisis

Hay diferentes formas de analizar el resultado del testing, dependiendo cuál sea el output de la aplicación. Por ejemplo: si el output es una tabla de resultados, el análisis de los resultados puede ser un script que realice la comparación de los datos resultantes contra los datos esperados. Debemos estudiar entonces si los criterios de cubrimiento han sido satisfechos.

4 Herramientas

Presentaremos ahora el conjunto de herramientas que permitirán llevar a cabo la metodología de testing desarrollada en el capítulo precedente.

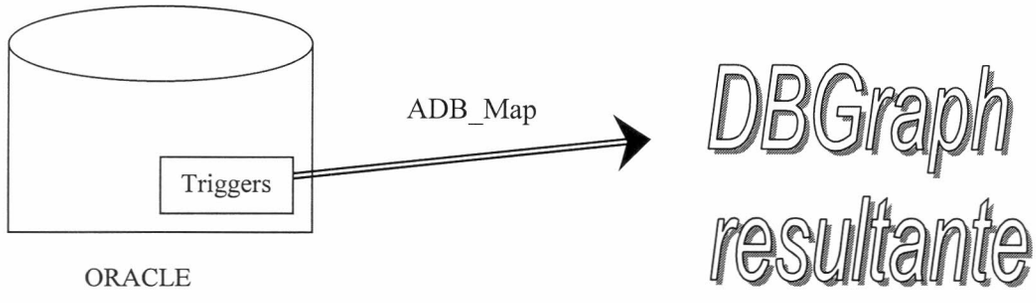
4.1 ADB_Map

La documentación técnica puede verse en el Anexo "ADB_Map".

4.1.1 Descripción

El ADB_Map es la herramienta que permite obtener al DBGraph relacionado a la base de datos activa que deseamos testear. Dicha herramienta se logró a través de un enfoque estático de la base de datos; se analiza el código de los triggers almacenados en la base de datos.

Gracias a un parser específico, se consiguió un mecanismo de abstracción que permite construir DBGraphs. El DBGraph nos permite observar la estructura de las reglas a testear.



El DBGraph es construído a partir del Diccionario de Datos de Oracle que almacena a las reglas de todos los usuarios. El estudio de las reglas puede realizarse ciertamente pues se almacena dentro de la base de datos al código fuente de las propias reglas.

El ADB_Map no es otra cosa que un parser específico que conoce la sintáxis del PL/SQL y de SQL. A este conocimiento hay que agregarle el de la relación entre la regla y su evento detonador. Esta relación también está establecida dentro de la base de datos.

No hemos tenido en cuenta las restricciones del trigger pues al realizar un estudio estático del código del mismo, no podemos determinar el valor lógico de la eventual

sentencia. En consecuencia, no modifica en lo mas mínimo al grafo resultante. Para mayores precisiones, remitirse a la sección 2.4.1.

Recordemos que el grafo resultante posee a todos los *posibles* ejes y nodos, luego en el momento de la ejecución efectiva de las reglas se sabe cuáles fueron recorridos. Esta característica se puede presentar en diversas y numerosas situaciones. Los casos mas frecuentes dónde se visualizan nodos y ejes inalcanzables son los siguientes:

- Condición de sentencia condicional de bifurcación falsa para cualquier conjunto de valores posible de entrada. Si bien esta situación es excluída como premisa de trabajo en muchas presentaciones teóricas, nosotros la tenemos en cuenta pues se da en el ámbito práctico que sirve de marco a este trabajo. Una instanciación posible sería:

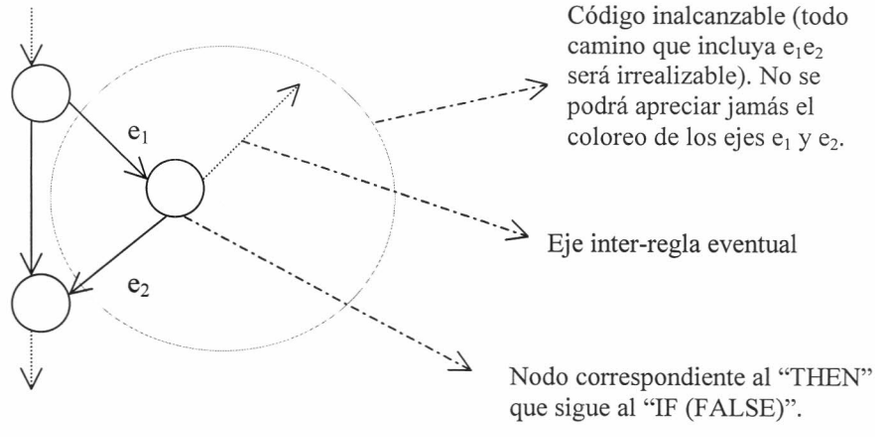
  ```
  IF (FALSE)
  THEN –
          (código PL SQL)
  END IF;
  ```

  La secuencia de sentencias anteriores dentro del cuerpo de un trigger generará dos ejes y un nodo que no serán nunca transitados, con la consecuente pérdida de calidad, riesgo de presencia de un error, etc …

- Una sentencia SQL dentro de un bloque de código inalcanzable. Esta situación se puede originar si en bloque compuesto por el código PL SQL del ejemplo anterior hubiese una sentencia de INSERT que detonase a otro trigger.

- Si hago "Update xxx set cccc where FALSE" y en la tabla xxx tengo un trigger para el update … no se detonará dicho trigger.

En las situaciones como las descritas anteriormente, se apreciará la utilización de esta herramienta pues se verá explícitamente (¡gráficamente!) la presencia de caminos no transitados. Si al querer transitarlos no lo logramos, habrá que sospechar de la alcanzabilidad de dichos caminos. Ejemplo:



Se realiza la inclusión de un único nodo inicial (R000) que se relaciona con los nodos iniciales de las reglas y de un único nodo final (R999) que se relaciona a su vez con los nodos terminales de las reglas. Esta inclusión de nodos ficticios nos permite tener un único punto de acceso y un único punto de salida en el grafo; y convertirlo entonces en un DBGraph.

El DBGraph resultante se presenta en un archivo.

4.1.2 Limitaciones

La versión desarrollada actualmente no posee:

Análisis completo del código de los "Triggers". Esto puede repercutir negativamente en diferentes circunstancias. Casos (entre otros) que pueden generar problemas son:

- Sentencias SQL preparadas en tiempo de ejecución no son tenidas en cuenta por el "parser" desarrollado.
- Sentencias SQL con condiciones incumplibles generan ejes imposibles de transitar.

Procesamiento de "Procedures" y "Functions". El código de estas funcionalidades provistas por Oracle no fue considerado dentro de nuestra herramienta. La utilización de dichas ventajas provistas puede modificar el grafo resultante pasando desapercibido por el ADB_Map. Para que el grafo resultante pierda un punto de contacto con la realidad, basta, por ejemplo, con que una *procedure* de una regla realice una acción sobre una tabla que detone a otra regla. En este caso la relación entre ambas reglas involucradas pasaría desapercibida al ADB_Map.

Utilización del SQL completo. Se ha restringido la sintáxis del SQL dejando de lado ciertas facilidades brindadas. Ver apéndice.

Alias para los campos referenciados. No son permitidos.

Parametrización de los mensajes de error. Cualquier mensaje de la herramienta que se desee modificar implica la recompilación de la misma. Para subsanar dichos problemas existen dos métodos principales: colocar a los mensajes de error dentro de una tabla de la base de datos o bien definirlos dentro de archivo único de parametrización.

4.2 ADB_Debugger

Ver documentación técnica en el Anexo "ADB_Debugger"

4.2.1 Descripción

La ejecución de este proceso prepara la Base de Datos para la realización del Test.

Esta herramienta modifica la Base de Datos Activa para que genere una traza recuperable. La forma en que logra su cometido es ubicando en "los lugares clave" del código un comando SQL de inserción. Dicho comando (un "INSERT") es inerte hacia el resto de la

aplicación a testear (para no generar distorsiones en nuestro test) pero no hacia el resto de las herramientas (en particular para el ADB_Tracer). El SQL inerte es una inserción de un registro (una traza) en una tabla privada a la herramienta para su posterior recuperación. Dicho registro está compuesto por el identificador del nodo en el cual se encuentra la sentencia de INSERT y un número de orden.

El problema principal para la construcción de esta herramienta consiste en conocer el lugar correcto para la localización de la traza (es decir, en que lugar del fuente colocar la sentencia INSERT). Es fundamental que la traza se refiera al mismo grafo generado por el ADB_Map. Esto posibilitará que luego de los casos de test se puedan extraer conclusiones que posean como base al estudio llevado a cabo sobre el grafo resultante del ADB_Map.

Sin embargo tras un análisis del problema llegamos a la conclusión que la mejor manera de asegurarnos que la concordancia buscada se iba a cumplir, era la utilización del mismo parser que se había utilizado para la herramienta ADB_Map en la herramienta ADB_Debugger.

¿Uno se podría preguntar si al agregar comandos dentro del código a testear, no vemos distorsionado el testing? Esta pregunta, propia a las técnicas de testing intrusivas, queda sin respuesta en este trabajo.

4.2.2 Limitaciones

La versión desarrollada actualmente:

Pasa todo el código a mayúsculas. Al realizar la inserción de las sentencias utilizadas para la traza, se devuelve el código fuente en mayúsculas.

No posee reconocedor de doble ejecución. Si se ejecuta dos veces el ADB_Debugger sin ejecutar entre medio al ADB_Clean, se realizará una doble inserción de las sentencias utilizadas para la traza.

4.3 ADB_Tracer

Ver documentación técnica en el Anexo "ADB_Tracer"

4.3.1 Descripción

Mediante la recuperación del subconjunto del DBGraph efectivamente testeado, se logra observar a la secuencia de ejecución de las reglas de la base de datos. La información se presenta coloreando a los ejes efectivamente transitados del DBGraph producido por el ADB_Map. Obtendremos en consecuencia un nuevo DBGraph que diferirá del original por el color de ciertos ejes.

Esta herramienta trabaja en conjunción al ADB_Debugger recuperando las trazas de las tablas internas a la herramienta.

Cabe aclarar que las trazas que se recuperan no son insertadas por el ADB_Debugger sino que son insertadas por el código a testear en el momento de su ejecución.

4.3.2 Limitaciones

La versión desarrollada actualmente no posee:

Diferenciación de Tests. No se pueden diferenciar los diferentes lotes de prueba, en consecuencia todos los casos de test ejecutados, pasarán a formar parte de un mismo lote único.

4.4 ADB_Cleaner

Ver documentación técnica en el Anexo "ADB_Cleaner"

4.4.1 Descripción

El ADB_Cleaner deja a la BDA en el mismo estado en el que se encontraba antes del inicio del Test.

La herramienta en cuestión retira del código fuente de las reglas a las sentencias SQL insertadas por el ADB_Debugger.

4.5 GraphViz

Se incorporó este producto luego de la evaluación de las siguientes alternativas:

1. Desarrollar una herramienta acorde a lo esperado.
2. Visualizar a los grafos mediante una herramienta no específica pero de amplia difusión.
3. Buscar y utilizar una herramienta existente que sea completamente acorde a lo esperado.

Hemos elegido a la última de estas alternativas, pues quedamos muy satisfechos frente al GraphViz.

4.5.1 Descripción

GraphViz es una herramienta de representación de grafos de dominio público desarrollada por AT&T que se encuentra disponible para una amplia gama de plataformas.

Si bien GraphViz cubre una amplia gama de funcionalidades, nos limitaremos a aprovechar sus excelentes aptitudes para la graficación de los grafos obtenidos sobre MS-Windows (3.1, 95, 98, NT).

El manual de la herramienta, así como la información necesaria para su instalación, los fuentes y todo tipo de información relacionada se puede encontrar en la Web de AT&T (http://www.research.att.com/sw/tools/graphviz/).

La interfase gráfica de GraphViz se guía a través de los siguientes principios para la visualización de los grafos:

- Brindar una clara legibilidad para el reconocimiento de los objetos a representar.
- Evitar confusiones producidas por el cruce de ejes y la superposición de objetos.
- Respetar posicionamientos intuitivos de los objetos a representar (minimizando tamaño de los ejes, posiciones relativas entre los nodos, etc.).
- Poner en relieve ciertos patrones memotécnicos (simetrías, paralelismos, regularidades).

Dentro de las facilidades ofrecidas por el producto, se encuentran las siguientes:

- Imprimir el gráfico resultante
- Funciones de "zoom".
- Modificar al grafo sobre el gráfico mismo sin necesidad de pasar a través del archivo que lo originó (i.e. agregar/eliminar/mover ejes y/o nodos sin modificar las especificaciones así como modificar atributos de los objetos del grafo como ser el color, borde, etiqueta, etc).
- Generar las especificaciones del grafo que se modificó.

La herramienta en cuestión requiere un archivo plano de entrada donde se encuentre especificado el grafo a representar. Obviamente, la especificación respeta una sintáxis determinada pero muy flexible.

La sintáxis utilizada por la herramienta en cuestión es la siguiente:

digraph *Nombre_Grafo* {
    Nombre_Nodo_Origen1 -> Nombre_Nodo_Destino1
    ...
    Nombre_Nodo_OrigenN -> Nombre_Nodo_DestinoM
}

dónde:
los identificadores con prefijo *Nombre_* pueden ser cualquier cadena de caracteres;
los puntos suspensivos (...) indican la posibilidad de tener varias líneas.

Diversas posibilidades, incluyendo el enriquecimiento de la sintáxis presentada están desarrolladas en el manual del producto.

5 Casos de estudio

En este capítulo presentamos tres casos de estudio donde hemos aplicado nuestra metodología de testing con la ayuda de las herramientas desarrolladas. El primero de ellos es el Sistema María. El interés de este caso es el de tratarse de un sistema de gran envergadura actualmente operativo desarrollado en la década de los '90. Intentaremos inferir a través del Sistema María cuál es la utilidad de nuestro trabajo en los usos de la industria local. El segundo caso es puramente teórico, creado por nosotros, e intenta estudiar nuestra metodología frente a una situación compleja. Por último, veremos un caso en el cual la utilización de la herramienta puede ocasionar distorsiones en el cubrimiento esperado.

5.1 Sistema María

El Sistema María es el resultado del proyecto de Informatización de la Aduana Argentina que relaciona a los agentes del Comercio exterior con los bancos, la Aduana, los agente de transporte, depositarios, etc ... El Sistema María se encuentra actualmente operativo en las aduanas más importantes del país.

5.1.1 Tablas involucradas y datos relevantes para el estudio de los triggers

Tabla de montos depositados por los despachantes de aduana en nombre de los importadores y/o exportadores: CMA

CMA	T	SUBCUENTA MARIA	
CCMABOL	C(9)	N	NUMERO DE MEDIO DE DEPOSITO
ICMAPAG	C(19)	N	IDENTIFICADOR DEL DEPOSITO
CCMADSP	C(11)	N	IDENTIFICADOR DEL DESPACHANTE
MCMAIMPPAG	N(18,2)	N	IMPORTE DEPOSITADO
DCMAACR	D(7)	N	FECHA DE ACREDITACION EFECTIVA
CCMAIMPEXP	C(11)	N	CUIT IMPORT / EXPORT
MCMASAL	N(18,2)	N	SALDO DISPONIBLE

Trigger: TR_MACMA
Criterio de aplicación: AFTER EACH ROW
Evento detonante: INSERT OR UPDATE OR DELETE
Tabla de acción: CMA
Descripción del código:

Este trigger registra toda modificación (en el sentido amplio, es decir ALTA + BAJA + MODIFICACION) realizada sobre las cuentas de los importadores/exportadores.
En el caso de un alta, sólo se registran las acreditaciones realizadas en una fecha diferente a la del día (para no sobrecargar a la auditoría interna).
En el caso de una modificación, sólo se registra a los movimientos que acrecienten al saldo de un operador.
Para las bajas, no se realizan contemplaciones pues se trata de un movimiento "prohibido".

Código:

```
begin
if INSERTING then

   if :new.dcmaacr != to_date(to_char(sysdate,'DD-MON-YY'),'DD-MON-YY')
    then
      insert into control.cmaaud (cmaaud_ccmabol, cmaaud_ccmabol_ant,
             cmaaud_icmapag, cmaaud_icmapag_ant,
             cmaaud_ccmadsp, cmaaud_ccmadsp_ant,
             cmaaud_ccmaimpexp, cmaaud_ccmaimpexp_ant,
             cmaaud_dcmaacr, cmaaud_dcmaacr_ant,
             cmaaud_mcmaimppag, cmaaud_mcmaimppag_ant,
             cmaaud_mcmasal, cmaaud_mcmasal_ant,
             cmaaud_user, cmaaud_useros,
             cmaaud_fecha)
      values (:new.ccmabol, null,
       :new.icmapag, null,
       :new.ccmadsp, null,
       :new.ccmaimpexp, null,
       :new.dcmaacr, null,
       :new.mcmaimppag, null,
       :new.mcmasal, null,
       user, user, sysdate);
      end if;
    else
   if UPDATING then
   if (
     (:old.ccmabol != :new.ccmabol)   or
     (:old.icmapag != :new.icmapag)   or
     (:old.ccmadsp != :new.ccmadsp)   or
     (:old.ccmaimpexp != :new.ccmaimpexp) or
     (:old.dcmaacr != :new.dcmaacr)   or
     (:old.mcmaimppag != :new.mcmaimppag) or
     (:new.mcmasal > :old.mcmasal)
    ) then
     insert into control.cmaaud (cmaaud_ccmabol, cmaaud_ccmabol_ant,
             cmaaud_icmapag, cmaaud_icmapag_ant,
             cmaaud_ccmadsp, cmaaud_ccmadsp_ant,
             cmaaud_ccmaimpexp, cmaaud_ccmaimpexp_ant,
             cmaaud_dcmaacr, cmaaud_dcmaacr_ant,
             cmaaud_mcmaimppag, cmaaud_mcmaimppag_ant,
             cmaaud_mcmasal, cmaaud_mcmasal_ant,
             cmaaud_user, cmaaud_useros,
             cmaaud_fecha)
     values (:new.ccmabol, :old.ccmabol,
       :new.icmapag, :old.icmapag,
       :new.ccmadsp, :old.ccmadsp,
       :new.ccmaimpexp, :old.ccmaimpexp,
       :new.dcmaacr, :old.dcmaacr,
       :new.mcmaimppag, :old.mcmaimppag,
       :new.mcmasal, :old.mcmasal,
       user, user, sysdate);
   end if;
   else
   insert into control.cmaaud (cmaaud_ccmabol, cmaaud_ccmabol_ant,
             cmaaud_icmapag, cmaaud_icmapag_ant,
             cmaaud_ccmadsp, cmaaud_ccmadsp_ant,
             cmaaud_ccmaimpexp, cmaaud_ccmaimpexp_ant,
             cmaaud_dcmaacr, cmaaud_dcmaacr_ant,
             cmaaud_mcmaimppag, cmaaud_mcmaimppag_ant,
             cmaaud_mcmasal, cmaaud_mcmasal_ant,
             cmaaud_user, cmaaud_useros,
             cmaaud_fecha)
   values (null, :old.ccmabol,
       null, :old.icmapag,
       null, :old.ccmadsp,
       null, :old.ccmaimpexp,
       null, :old.dcmaacr,
```

```
        null, :old.mcmaimppag,
        null, :old.mcmasal,
        user, user, sysdate);
    end if;
  end if;
   end;
```

Tabla de devoluciones tributarias (montos que percibirán los importadores y/o exportadores a raíz de estímulos a la exportación, reintegros, etc …): DEO

DEO	T DEVOLUCION		
IDEO	C(12)	N	IDENTIFICADOR DE LA DEVOLUCION
CDEODSP	C(11)		IDENTIFICADOR DEL DESPACHANTE
MDEOIMPD	N(18,2)	N	IMPORTE DE LA DEVOLUCION
CDEOIMPEXP	C(11)		CUIT DEL EXPORTADOR/IMPORTADOR

Trigger: TR_MBDEO
Criterio de aplicación: BEFORE EACH ROW
Evento detonante: UPDATE OR DELETE
Tabla de aplicación: DEO
Descripción del código:

El trigger prohíbe todo intento de modificación de la clave de la devolución tributaria o borrado de la misma.

Código:

```
begin
  if UPDATING then
    if (:old.ideo != :new.ideo) then
      raise_application_error(-20000,'Error al modificar DEO');
    end if;
  else
    raise_application_error(-20000,'Error al borrar DEO');
  end if;
end;
```

Tabla de bancos: BNQ

BNQ	T BANCO

Trigger: TR_MBBNQ
Criterio de aplicación: BEFORE EACH ROW
Evento detonante: INSERT OR UPDATE OR DELETE
Tabla de aplicación: BNQ
Descripción del trigger:

No están permitidas las modificaciones sobre los atributos de los bancos entre las 21:00 y las 24:00.

Código:

```
declare
  f_com  date := trunc(sysdate)+(20.5/24);
  f_fin  date := trunc(sysdate)+1;
```

```
begin
  if sysdate between f_com and f_fin then

    raise_application_error(-20001,'Transaccion no disponible de ' ||
                            to_char(f_com,'HH24:MI') || ' a '      ||
                            to_char(f_fin,'HH24:MI')) ;
  end if;
end;
```

Tabla de registro de importadores y exportadores: IPR

IPR	T	IMPORTADOR/EXPORTADOR
CIPR	C(11)	N IDENTIFICADOR DEL IMPORTADOR/EXPORTADOR
LIPRNOM	C(30)	N NOMBRE DEL IMPORTADOR/EXPORTADOR
DIPREFF	(7)	N FECHA DE INICIO DE VIGENCIA
DIPRFIN	(7)	N FECHA DE FIN DE VIGENCIA

Trigger: IPR1TR
Criterio de aplicación: BEFORE EACH ROW
Evento detonante: INSERT OR UPDATE
Tabla de aplicación: IPR
Descripción del código:

No se permite la existencia de dos importadores/exportadores con la misma CUIT.

Código:

```
declare
   x              CHAR(1);
   IPR_DUPLICADO  EXCEPTION;
begin
   if INSERTING then
      select 'x' into x
        from ipr
       where cipr = :new.cipr
         and (dipreff <= :new.dipreff and :new.dipreff < diprfin or
              dipreff < :new.diprfin and :new.diprfin <= diprfin or
              :new.dipreff <= dipreff and dipreff < :new.diprfin or
              diprfin > :new.dipreff and diprfin <= :new.diprfin)
         and rownum < 2;
      raise IPR_DUPLICADO;
   elsif UPDATING then
     select 'x' into x
       from ipr1_v
       where cipr = :new.cipr
         and (dipreff <= :new.dipreff and :new.dipreff < diprfin or
              dipreff < :new.diprfin and :new.diprfin <= diprfin or
              :new.dipreff <= dipreff and dipreff < :new.diprfin or
              diprfin > :new.dipreff and diprfin <= :new.diprfin)
         and dipreff != :old.dipreff
         and diprfin != :old.diprfin
         and rownum < 2;
     raise IPR_DUPLICADO;
   end if;
exception
   when IPR_DUPLICADO then
     raise_application_error(-20000, 'El lapso de habilitacisn del
importador '
                             || :new.cipr || ' se superpone con uno
vigente.');
   when NO_DATA_FOUND then
```

```
        null;
end;
```

Tabla de compañías de seguro: SEG

SEG	T COMPANIA DE SEGUROS

Trigger: TR_MBSEG
Criterio de aplicación: BEFORE EACH ROW
Evento detonante: INSERT OR UPDATE OR DELETE
Tabla de aplicación: SEG
Descripción del código:

No están permitidas las modificaciones sobre los atributos de las compañias de seguros entre las 21:00 y las 24:00.

Código:

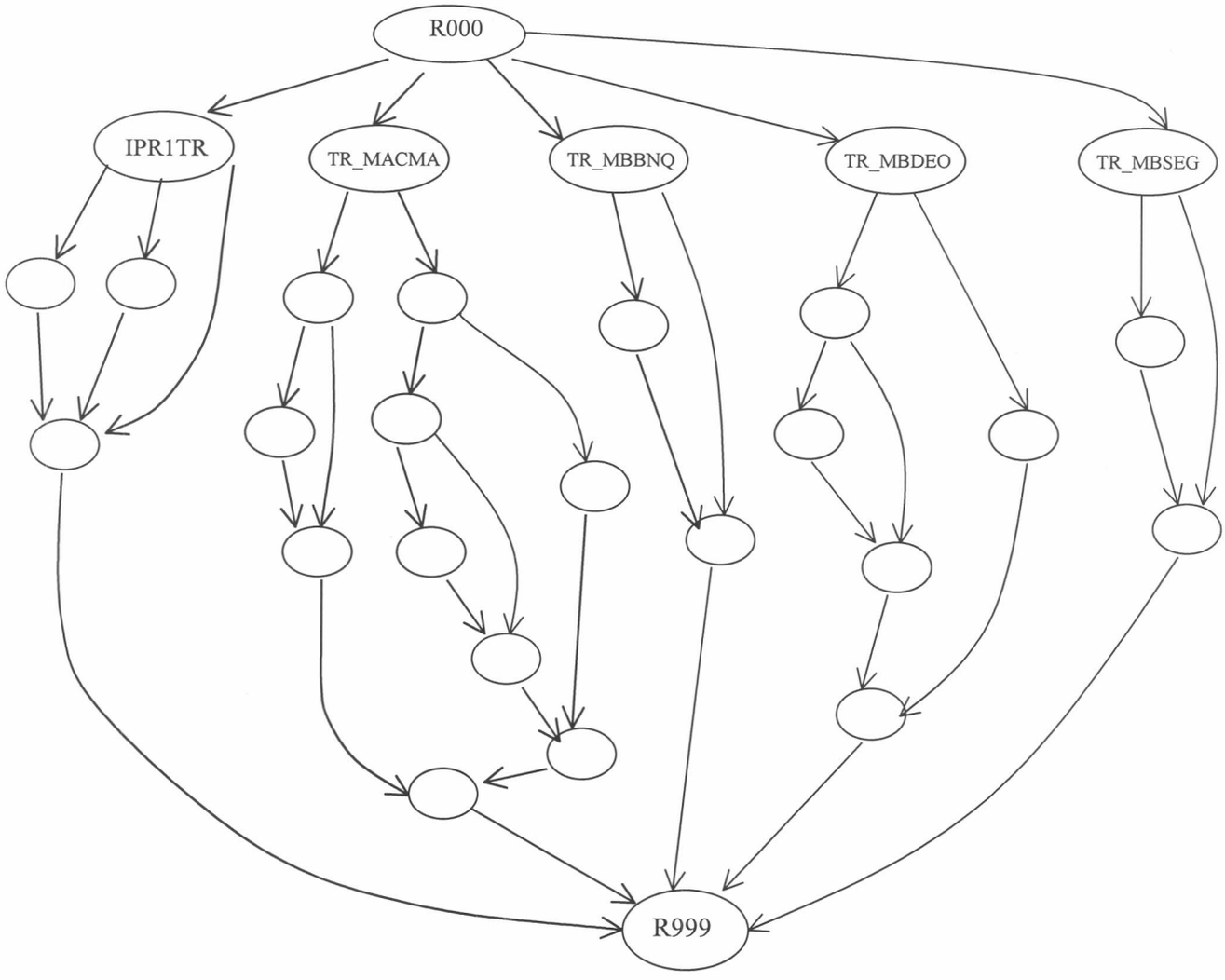
```
declare
  f_com  date := trunc(sysdate)+(20.5/24);
  f_fin  date := trunc(sysdate)+1;
begin

  if sysdate between f_com and f_fin then
    raise_application_error(-20001,'Transaccion no disponible de ' ||
                            to_char(f_com,'HH24:MI') || ' a '      ||
                            to_char(f_fin,'HH24:MI')) ;
  end if;
end;
```

Grafo asociado al esquema presentado:

```
digraph BDA {
 R000 -> IPR1TR
 R000 -> TR_MACMA
 R000 -> TR_MBBNQ
 R000 -> TR_MBDEO
 R000 -> TR_MBSEG
 IPR1TR -> IPR1TR_5
 IPR1TR -> IPR1TR_7
 IPR1TR -> IPR1TR_6
 IPR1TR_5 -> IPR1TR_6
 IPR1TR_7 -> IPR1TR_6
 TR_MACMA -> TR_MACMA_9
 TR_MACMA -> TR_MACMA_13
 TR_MACMA_9 -> TR_MACMA_9_10
 TR_MACMA_9 -> TR_MACMA_9_11
 TR_MACMA_9_10 -> TR_MACMA_9_11
 TR_MACMA_9_11 -> TR_MACMA_12
 TR_MACMA_13 -> TR_MACMA_13_14
 TR_MACMA_13 -> TR_MACMA_13_18
 TR_MACMA_13_14 -> TR_MACMA_13_14_15
 TR_MACMA_13_14 -> TR_MACMA_13_14_16
 TR_MACMA_13_14_15 -> TR_MACMA_13_14_16
 TR_MACMA_13_14_16 -> TR_MACMA_13_17
 TR_MACMA_13_17 -> TR_MACMA_12
 TR_MACMA_13_18 -> TR_MACMA_13_17
```

```
TR_MBBNQ -> TR_MBBNQ_20
TR_MBBNQ -> TR_MBBNQ_21
TR_MBBNQ_20 -> TR_MBBNQ_21
TR_MBDEO -> TR_MBDEO_23
TR_MBDEO -> TR_MBDEO_27
TR_MBDEO_23 -> TR_MBDEO_23_24
TR_MBDEO_23 -> TR_MBDEO_23_25
TR_MBDEO_23_24 -> TR_MBDEO_23_25
TR_MBDEO_23_25 -> TR_MBDEO_26
TR_MBDEO_27 -> TR_MBDEO_26
TR_MBSEG -> TR_MBSEG_29
TR_MBSEG -> TR_MBSEG_30
TR_MBSEG_29 -> TR_MBSEG_30
IPR1TR_6 -> R999
TR_MACMA_12 -> R999
TR_MBBNQ_21 -> R999
TR_MBDEO_26 -> R999
TR_MBSEG_30 -> R999
}
```

dicho grafo se presenta a través del Graphviz (obviamos en los gráficos siguientes la mayoría de los nombres de los nodos para lograr legibilidad)…



5.1.2 Testing

Podríamos aplicar diferentes criterios de adecuación del testing realizado para evaluar los resultados obtenidos e intentar la extracción de alguna conclusión. Los criterios evaluados son:

1. Criterio "Cubrir todos los ejes disparadores"
2. Criterio "Cubrir todos los ejes"
3. Criterio "Cubrir todos los nodos"
4. Criterio "Cubrir ejes iniciales y disparadores"

Desarrollaremos los inputs necesarios para satisfacer el cuarto criterio presentado "Cubrir todos los ejes iniciales y disparadores".

Como ninguna regla puede detonar a otra, la forma de asegurarnos la métrica propuesta es simplemente detonar a cada una de las reglas. Para detonar a las reglas involucradas podemos realizar las siguientes operaciones:

a) Depositar un monto en la cuenta Maria de un importador

```
INSERT INTO  CMA ()
VALUES       ('1', '123456789', '301234567891',
                    2000, '08-jul-98', '309874561230', 3000);
```

b) Borrar una devolución tributaria

```
DELETE       DEO
WHERE        rownum        <        2;
```

c) Eliminar al úlitmo banco insertado

```
DELETE       BNQ
WHERE        rowid  =      (SELECT       max(rowid)
                    FROM         BNQ);
```

d) Darme de alta como importador

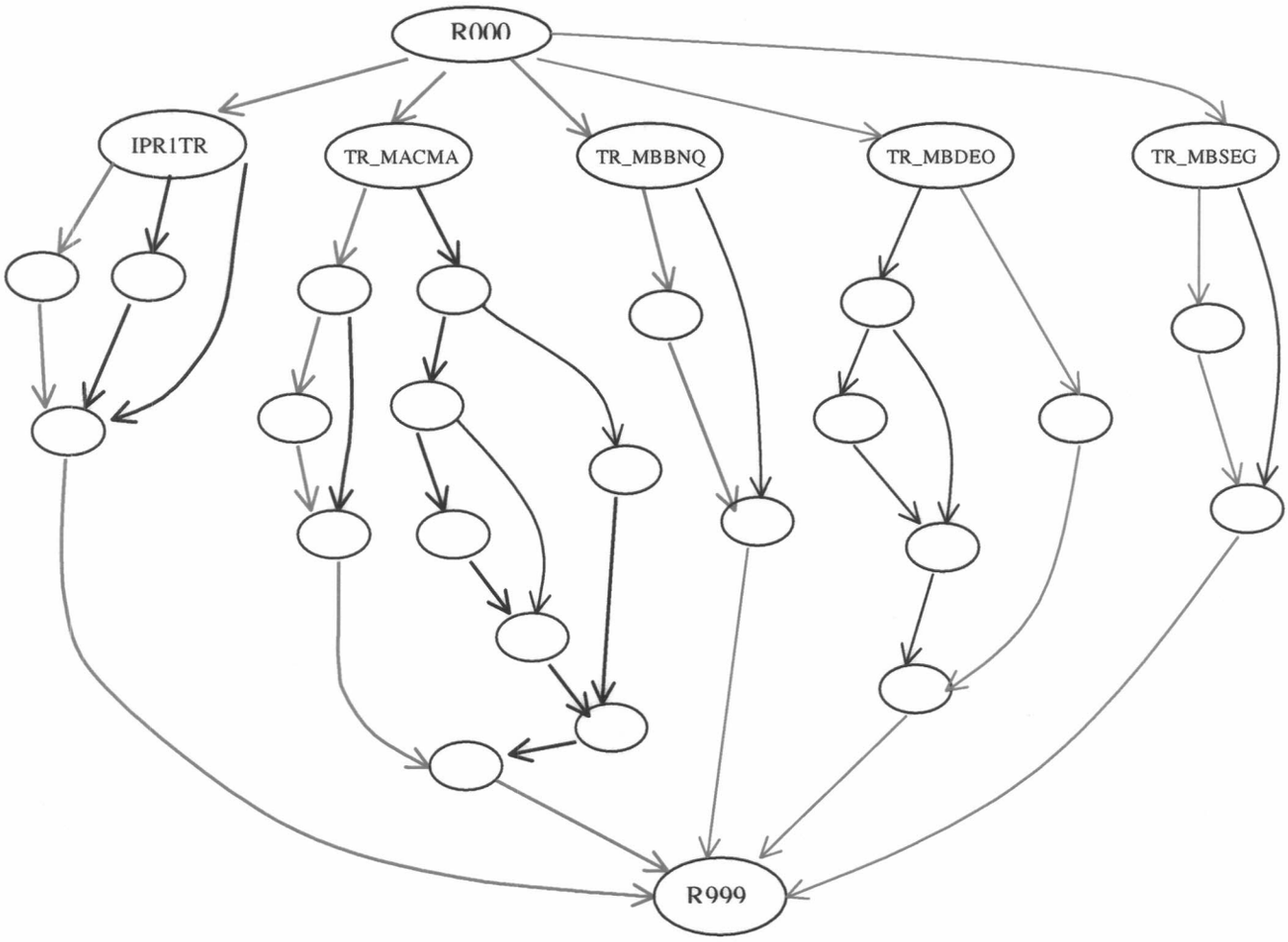
```
INSERT INTO  IPR
VALUES       (…);
```

e) Eliminar a la última compañia de seguros insertada

```
DELETE       SEG
WHERE        rowid  =      (SELECT       max(rowid)
                    FROM         SEG);
```

5.1.3 Resultado esperado

El resultado esperado lo podemos expresar a través del grafo con la cobertura del DBGraph esperada:

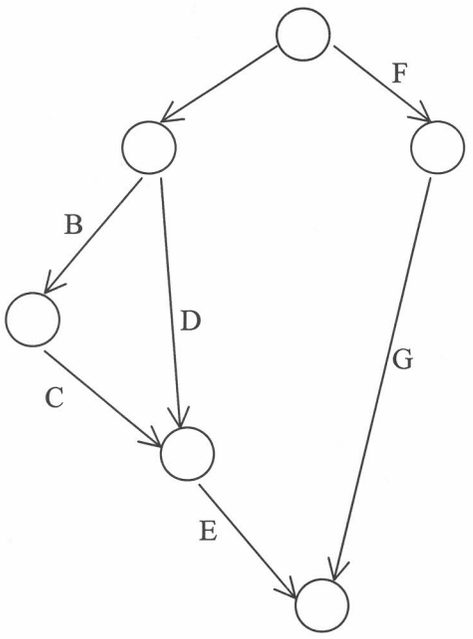


Vemos en color rojo, a los ejes que esperamos recorrer gracias al test a realizar.

Aclaración:

con la mayoría de los cubrimientos estudiados no podemos predecir con certeza el o los caminos recorridos pues podemos tener diferentes caminos con el mismo resultado final. El grafo resultante lo podemos predecir una vez que nos hallamos planteado los casos de test a ejecutar.

Podemos ver este planteo en el DBGraph del siguiente ejemplo (ver figura mas abajo). En este caso, si intento cubrir sólo los ejes iniciales, tengo dos conjuntos de caminos completos que me brindan dicho cubrimiento: {ABCE, FG} y {ADE, FG}. Cualquier de estos dos cubrimientos me satisface ya que contienen al eje A y al eje F (A y F son los únicos ejes iniciales). Sin embargo tengo un problema si quiero predecir el cubrimiento esperado pues debería analizar los casos de test para determinar cuál de los dos se efectuará.



5.1.4 Resultado efectivo del test

Debemos buscar siempre una corroboración fáctica de los hechos predichos, y es así como para el ejemplo dado, obtendremos exactamente el mismo grafo que en el punto anterior.

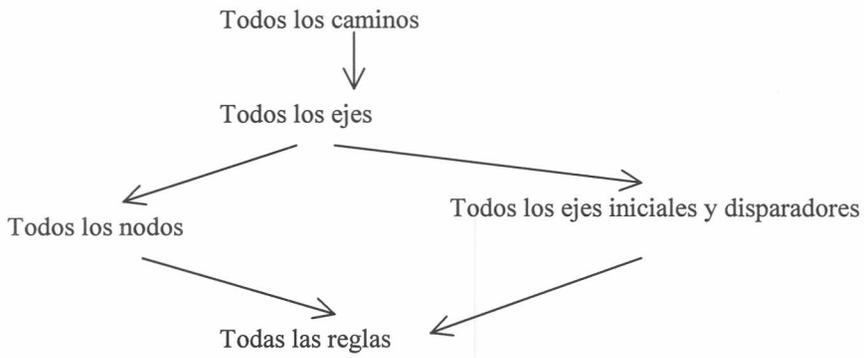
5.1.5 Análisis de resultados

Este ejemplo nos permite observar y analizar diferentes cosas, a saber:

(a) Relación con los resultados teóricos sobre cubrimientos desarrollados en [Bal97]
(b) Relaciones comparativas sobre cubrimientos

(a)

La tesis citada definió un número de cubrimientos, una relación de orden entre dos cubrimientos y luego ordenó a los cubrimientos definidos. El resultado que nos interesa de dicho estudio es el siguiente:



En el marco de esta tesis se modificó la representación de una regla (se redefinió su grafo, ver sección 2.2) y se debieron redefinir a los DBGraphs (por el modelo de reglas que implementó Oracle). En consecuencia se podría estudiar si se han conservado las relaciones de orden establecidas para los diferentes cubrimientos.

Nosotros nos limitaremos a verificar si los ejemplos dados han respetado el orden definido en [Bal97] o bien si lo han vulnerado.

Recordemos los criterios estudiados en este ejemplo:

1. Criterio "Cubrir todos los ejes disparadores"
2. Criterio "Cubrir todos los ejes"
3. Criterio "Cubrir todos los nodos"
4. Criterio "Cubrir ejes iniciales y disparadores"

Utilizaremos el símbolo "☐" para relacionar dos criterios con el siguiente significado: "a ☐ b, si al respetar el criterio b, respetamos también al criterio a".

La relación de orden observada en este ejemplo es la siguiente:

1 ☐ 4, 4 ☐ 2, 1 ☐ 2, 4 ☐ 3, 1 ☐ 3 y 2 ☐ 3

Sobre estas relaciones podemos afirmar que se ratificaron las siguientes relaciones teóricas:

1 ☐ 4, 4 ☐ 2, 1 ☐ 2 y 2 ☐ 3

Sin embargo, deberíamos estudiar que 3 es comparable a 1 y a 4, pues esto contradiría lo estudiado en [Bal97]. El motivo que nos permite comparar ambos criterios en este caso de estudio puntual, se origina en la inexistencia de ejes inter-reglas.

(b)

La relación "☐" utilizada permite ordenar a los diferentes criterios pero sin embargo no nos sirve para cuantificar que tan cercanos o alejados se encuentran dos criterios. Si queremos decidir entre dos criterios a utilizar, nos pueden ser útiles las jerarquías teóricas que relacionan a los diferentes criterios. Ahora bien, si queremos adoptar un punto de vista un poco mas práctico para decidir que criterio utilizar de los disponibles, querríamos conocer también "que tanto ganamos y que tanto perdemos" al tomar una elección.

Necesitaríamos "enriquecer" a la relación "□" para poder cuantificar de alguna manera que tan mayor es un criterio con relación a otro.

El significado dado a la relación "□" determina el análisis que podemos realizar, y en este caso nos tenemos que limitar a las relaciones observadas en 5.1.5 (a).

Para el caso de estudio presentado alcanzar los criterios 1) y 4) resulta "barato" en comparación a los otros dos criterios propuestos.

Tomemos como ejemplo el caso del eje que une a los nodos TR_MBBNQ y TR_MBBNQ_21 que sólo podrá ser recorrido si efectuamos una acción sobre la tabla BNQ entre las 21:00 horas y las 24:00 horas. Esto sucede pues el código del trigger TR_MBBNQ está compuesto por una sentencia alcanzable sólo en el caso de encontrarse su ejecución dentro del rango horario mencionado.

Se requerirá un esfuerzo de logística (para disponer de recursos en un horario no estándar) o coordinación (para cambiarle la hora al servidor de la aplicación) que aumenta la complejidad del test a realizar. Si nos guiamos por el criterio 1) o 4) dentro del horario laboral mas habitual en Argentina (9h a 18h) como parámetro de cumplimiento del testing, no habremos testeado una parte fundamental del trigger TR_MBBNQ (la parte que protege la información relativa a los bancos en el horario reservado a las transferencias).

En consecuencia, no podemos darnos por satisfechos con las pruebas realizadas pues no hemos verificado partes importantes de código en diferentes reglas (ver el caso de TR_MACMA por ejemplo!).

Esta situación no deseable se hubiese podido prevenir de tener una jerarquía entre criterios "mas rica" con la cual decidir el testing a realizar.

5.2 Caso teórico

Vamos a basarnos en un ejemplo desarrollado en la tesis de Martín Balzamo para construir otro caso de estudio.

Tablas presentes en el ejemplo

- EMP (id, rank, salary), tabla que almacena a los empleados junto a sus rangos y salarios correspondientes.
- BONUS (emp_id, amount), tabla que almacena la cantidad de bonus a percibir por cada empleado.
- SALES (emp_id, month, number), tabla que almacena la cantidad de ventas efectuadas por cada empleado en cada mes.
- CLIENT (emp_id, client_id), tabla que relaciona a cada empleado con sus clientes.
- VACACION (emp_id, cant), tabla que contabiliza la cantidad de días de vacaciones de cada uno de los empleados.

Triggers presentes

- Trigger "Good-Sales"

Criterio de aplicación: AFTER EACH ROW
Evento detonante: INSERT
Tabla de aplicación: SALES
Descripción del código:

Aumenta el salario de un empleado en 10 $ cada vez que éste vende en una misma operación mas de 50 unidades.
Aumenta el rango de un empleado en 1 cuando éste realiza una venta de mas de 100 unidades.

Código:

```
Begin
  --
  if new.number > 50
  then -
     update emp
     set emp.salary = emp.salary + 10
     where emp.id = new.emp_id;
  end if;
  --
  if new.number > 100
  then -
     update emp
     set emp.rank = emp.rank + 1
     where emp.id = new.emp_id;
  end if;
  --
end;
```

- Trigger "Rank-Raise"

Criterio de aplicación: BEFORE EACH ROW
Evento detonante: UPDATE(rank)
Tabla de aplicación: EMP
Descripción del código:

Aumenta el salario de un empleado en un 10% cada vez que su rango llega a 15.
Disminuye el salario en 5$ cada vez que se pierde categoría.

Código:

```
Begin
  --
  if new.rank = 15 and new.rank > old.rank
  then -
     update emp
     set emp.salary = emp.salary * 1.1
     where emp.id = new.emp_id;
  end if;
  --
  if new.rank < old.rank
  then -
     new.salary = new.salary - 5;
  end if;
  --
end;
```

- Trigger "Ethic Employee"

Criterio de aplicación: BEFORE EACH ROW
Evento detonante: INSERT
Tabla de aplicación: CLIENT
Descripción del código:

Penaliza el hecho de relacionar un mismo cliente a dos empleados diferentes. La forma de controlarlo elegida es la siguiente: cada vez que se da de alta una relación en la tabla CLIENT se chequea la cantidad de relaciones que posee el cliente a insertar en la tabla CLIENT, si ya se cuenta con una entrada en esta última tabla se penaliza al empleado en cuestión disminuyéndole el rango. Esta acción detona a su vez a otra regla ("Rank Raise") que le bajará automáticamente el sueldo.
Aumenta el salario de un empleado en 10$ por cada cliente nuevo por sobre los primeros 20 clientes.

Código:

```
Declare
  cuantos     number;
begin
  --
  select count(*)
  into cuantos
  from client
  where new.emp_id != client.emp_id and
        new.client_id = client.client_id;
  --
  if cuantos > 0
  then -
     update emp
     set rank = rank -1
     where emp.id = new.emp_id;
  end if;
  --
```

```
  select count(*)
  into cuantos
  from client
  where new.emp_id = client.emp_id;
  --
  if cuantos > 20
  then -
     update emp
     set emp.salary = emp.salary + 10
     where emp.emp_id = new.emp_id;
  end if;
  --
end;
```

- Trigger "Get Bonus"

Criterio de aplicación: AFTER EACH ROW
Evento detonante: UPDATE(salary)
Tabla de aplicación: EMP
Descripción del código:

Aumenta en un 5 por ciento los bonos a cobrar por un empleado que haya sido beneficiado con un aumento salarial superior al 50 por ciento.

Código:

```
begin
  IF :new.salario > :old.salario * 1.5
  THEN --
      UPDATE bonus
      SET bonus.monto = bonus.monto * 1.05
      WHERE bonus.ident = :new.ident;
  END IF;
end;
```

- Trigger "Great Bonus"

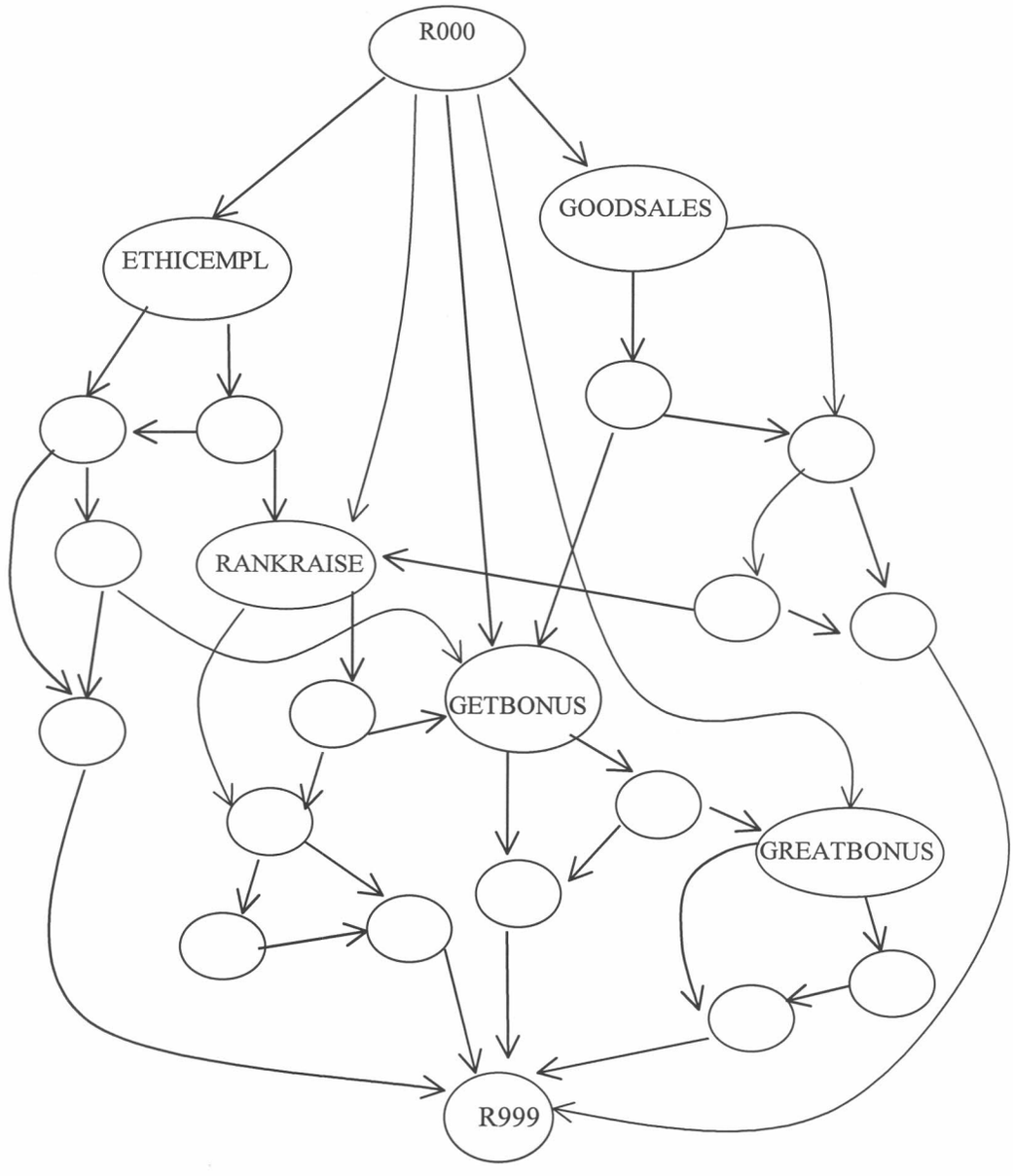
Criterio de aplicación: AFTER EACH ROW
Evento detonante: INSERT OR UPDATE
Tabla de aplicación: BONUS
Descripción del código:

Aumenta en 5 la cantidad de días de vacaciones disponibles por un empleado que haya sido beneficiado con un aumento de bonos superior al 50 por ciento.

Código:

```
begin
  IF :new.monto > :old.monto * 1.5
  THEN --
      UPDATE vacacion
      SET vacacion.cant = vacacion.cant + 5
      WHERE vacacion.ident = :new.ident;
  END IF;
end;
```

Grafo asociado



Fijarse un criterio para el cubrimiento a alcanzar

Volvemos a ver cuáles son los criterios de adecuación del testing realizado más sencillos de utilizar:

1. Criterio "Cubrir todos los ejes disparadores"
2. Criterio "Cubrir todos los ejes"
3. Criterio "Cubrir todos los nodos"
4. Criterio "Cubrir ejes iniciales y disparadores"

A pesar de los resultados obtenidos en [Bal97], nosotros compararemos los criterios 3 y 4 (como fue visto en la discusión de la sección 5.1.5).

El criterio 4 no difiere demasiado con el 3, de hecho la sola posible diferencia se da en el cubrimiento o no de los nodos RANKRAISE_19 y GREATBONUS_14 (obviamente, la métrica 3 es mas segura que la 4). Estos dos criterios nos permiten cubrir un gran porcentaje de caminos posibles sobre el DBGraph de base.

Sabemos (ver tesis [Bal97]) que el criterio 2 es el que mayores garantías nos brinda, sin embargo el costo de llevarla a cabo es mucho mayor a los otros dos.

Por último nos queda una observación para el criterio 1. Luego de una visión de lo que él representa sobre el caso de estudio en cuestión, nos damos cuenta que no presenta diferencias en cuanto a lo que cubre con respecto al criterio 4. La diferencia entre ambos es importante pues no poseen el mismo costo llevarlas a cabo. Mientras que para el criterio 4 hacían falta como mínimo 5 sentencias sql, para el criterio 1, bastan 2 sentencias sql …

Para no aburrir al lector, nos limitaremos entonces a llevar a cabo el primer criterio mencionado "Cubrir los ejes disparadores".

Realizar los tests

Sentencias SQL que posibilitan llevar a cabo el criterio propuesto, "Cubrir todos los ejes disparadores":

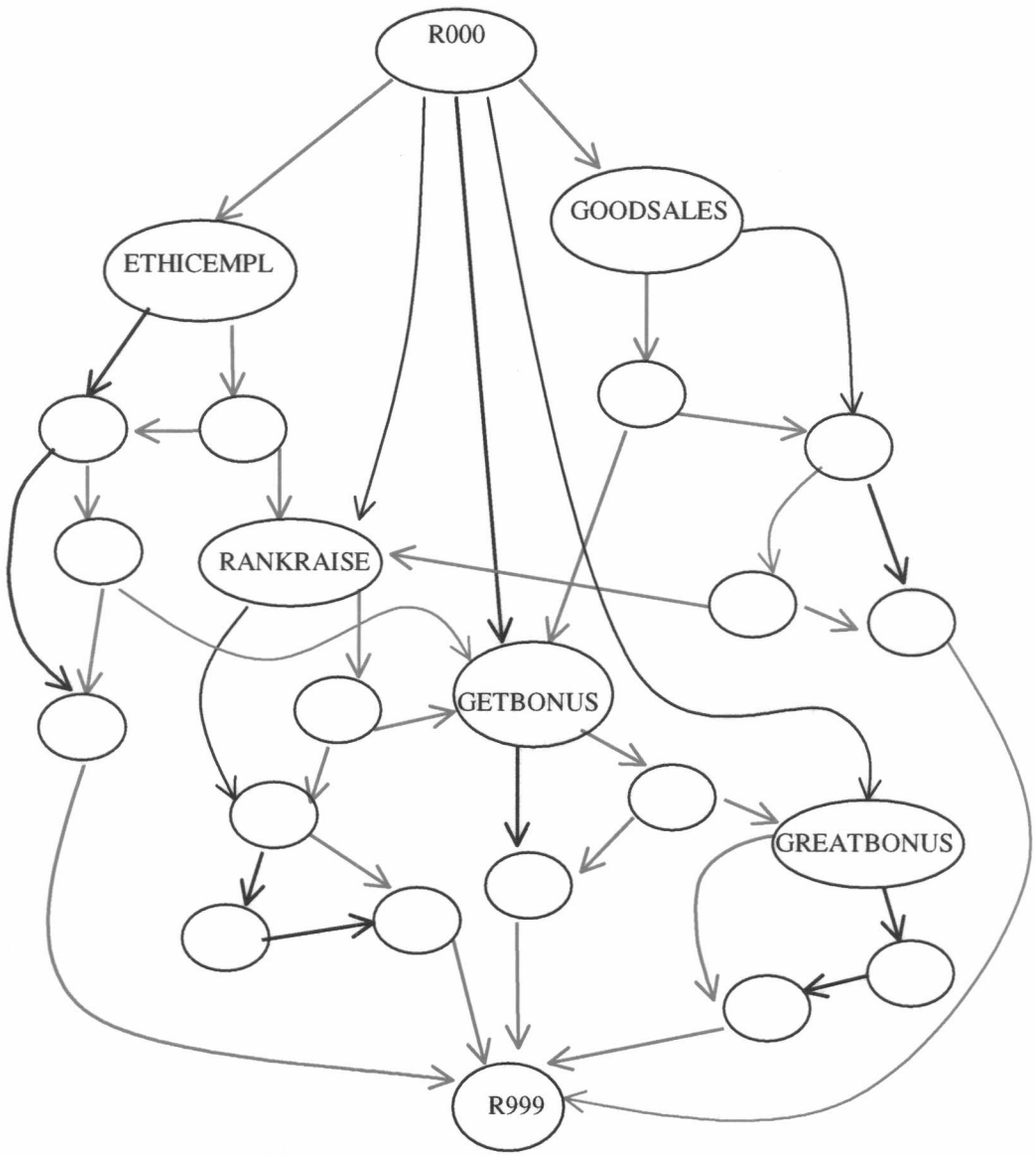
a) Insertar un cliente CA (que ya esté relacionado con un empleado, digamos EA) para un empleado EB que posea mas de 20 clientes en su cuenta.

```
INSERT INTO CLIENTE
VALUES ('EB', 'CA');
```

b) Insertar una venta de mas de 100 unidades a un empleado EA.

```
INSERT INTO SALES
VALUES ('EA', 09, 120);
```

Grafo resultante



Podemos observar en color rojo a los ejes transitados gracias a nuestros comandos SQL.

Observaciones

En el caso de estudio presente, al transitar los nodos como ETHICEMPL_1, estamos tomando la bifurcación creada para excepciones al uso general del programa (el nodo ETHICEMPL_1, se debería ejecutar frente a un empleado que le está quitando el cliente a un compañero de trabajo, situación que no debería ser demasiado corriente por el bien de la empresa ficticia …).

¿Cuál es la deficiencia de aplicar dicha cobertura en nuestro proceso de testing?
La deficiencia se encuentra, como a menudo en el testing estructural, enraigada en el desconocimiento funcional de la aplicación a probar.

En definitiva, testeamos un conjunto de caminos que si bien son topológicamente atractivos, no son los más usuales ni los mas importantes de testear en una primera instancia.

5.3 Base de datos distribuída

Se trata de un ejemplo ficticio que sirve como caso de estudio de un problema que se puede presentar, en consecuencia no explicitaré el desarrollo de las reglas presentes sino que me concentraré en la parte que interesa resaltar.

La problemática es la que se presenta a continuación. Tenemos una aplicación distribuida en dos servidores con sus respectivas bases de datos. La utilización de nuestra aplicación detonará reglas en ambas bases de datos y ciertas reglas de una base de datos detonan reglas de la otra.

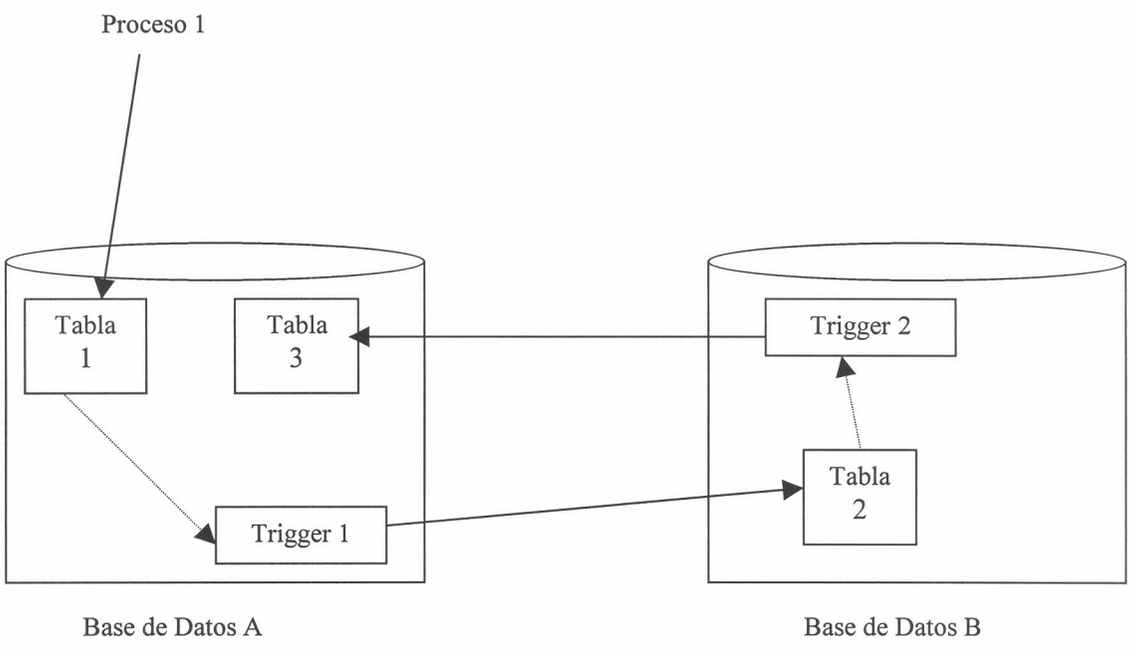


Figura "Base de datos distribuida"

En el esquema que presentamos en la figura "Base de datos distribuida", podemos observar el caso de estudio claramente. Un proceso (el proceso 1) realiza una operación sobre la tabla 1; dicha operación detona un trigger (el trigger 1) que realiza a su vez una operación sobre la tabla 2 que se encuentra en otra base de datos.

Al realizarse esta última operación sobre otra base de datos, es susceptible de detonarse otro conjunto de triggers que no habían estado contemplados en la base de datos A. En esta situación se encuentra el trigger 2, que se detonó producto de una acción realizada por el trigger 1.

Podemos imaginarnos tantos niveles de complejidad como queramos, como ser que la acción llevada a cabo por el trigger 2 a raíz de la acción llevada a cabo sobre la tabla 2

produzca como efecto colateral una nueva modificación en una tabla (la tabla 3) de la base de datos A.

Ahora bien, nuestro producto trabaja con una única base de datos sin procesar los "dblinks" eventuales. Y en consecuencia, la estructura de la base de datos activa obtenida a través de ADB_Map resultará incompleta pues será incapaz de mostrar las relaciones que existen entre las componentes de las diferentes bases de datos.

En el caso presentado en la figura mencionada, el ADB_Map no tendrá en cuenta a la base de datos B, ni tampoco al eje que relaciona al Trigger 1 con el Trigger 2, y en consecuencia el DBGraph obtenido no modelizará apropiadamente a la realidad.

6 Conclusiones y extensiones

6.1 Conclusiones y otras consideraciones

Así como las herramientas CASE (Computer Aided Software/Systems Engineering) constituyen un avance en la "industrialización" del análisis, diseño y construcción del soft, el ADB_Tester constituye un avance en el testeo de bases de datos activas.

Dado el objetivo a alcanzar (el testeo de una base de datos activa), el ADB_Tester no automatiza todas las etapas del testing sino que constituye una herramienta que facilita la realización del testing. Esta diferencia es importante pues determina el alcance de la herramienta.

El ADB_Tester requiere una injerencia humana en las diferentes fases del Test pero brindándole simultáneamente una ayuda importante. Como hemos visto en los diferentes ejemplos desarrollados anteriormente, esto resulta de vital importancia para poder determinar el tipo de testeo a realizar y, con posterioridad, saber interpretar los resultados obtenidos. Cuando hablamos de "ayuda", hablamos del incremento de la productividad y de la calidad del proceso de Test.

Agregamos en lo que respecta a la confiabilidad de los resultados a obtener la siguiente característica:

> Dado que nadie pretende del ADB_Tester una certeza de correctitud de la base de datos testeada, no se deben deslindar responsabilidades del Test volcándolas hacia la herramienta en cuestión. La herramienta desarrollada no realiza un testeo de la base de datos activa, sino que brinda una ayuda para la realización del testeo y la automatización del proceso de testing para el individuo responsable del testeo.

Podemos definir entonces al ADB_Tester como un conjunto de herramientas, técnicas (manuales y automatizadas) que permiten desarrollar testeos estáticos y dinámicos de una base de datos activa.

La conclusión más importante es, desgraciadamente, que el uso de bases de datos activas no forma parte de la cultura informática actual. Los casos reales relevados no usan demasiada complejidad en el diseño, en la estructura de las reglas y ni siquiera se trasladan complejas reglas de negocio en reglas de la base de datos. Esta información se ve avalada en la complejidad de los grafos resultantes luego de ejecutar el ADB_Map en una aplicación importante operativa.

- **¿Hasta que punto se debe involucrar al testeador en las especificaciones de la base de datos activa?**

Hay varios motivos para buscar un testeador no demasiado comprometido con las especificaciones a testear. Entre los mas importantes se podrían citar una mayor objetividad acerca de los resultados esperados y una mayor seguridad de que se realizará un conjunto de pruebas diferentes a las que realizó el equipo de desarrollo del producto a testear.

Sin embargo, hemos visto a través del ejemplo desarrollado en el caso de la base de datos distribuida (sección 5.3) que el testeador no puede alcanzar sus objetivos sin una inmersión dentro de las especificaciones técnicas del producto a testear. Si el testeador no es consciente del diseño de las reglas de la aplicación a testear, no podrá en este caso obtener total certeza de la correctitud de lo realizado.

Otro tipo de inmersión dentro de las especificaciones se requerirá para el testeo del "Caso Sistema MARIA". No será necesario involucrarse demasiado dentro de las especificaciones. Pero, como vimos dentro de las conclusiones del mismo (sección 5.1.5), hay ciertas restricciones funcionales como ser el horario que pueden entorpecer el proceso de test.

- **¿Cuál es la utilidad de la herramienta desarrollada?**

Constituye una importante ayuda tanto para el desarrollo de las pruebas de escritorio, como para las pruebas dinámicas. La ayuda fundamental radica en la visualización de la estructura del código de la base de datos activa y su consiguiente facilidad para una abstracción del tipo de código a testear. Dicho poder de abstracción resulta atractivo para estudiar código fuente complejo (con gran cantidad de iteraciones y/o bifurcaciones anidadas) o voluminoso (con gran cantidad de líneas de código).

La otra gran ayuda de la herramienta aparece cuando se decide la utilización de criterios de adecuación del testing realizado como los que hemos visto. En estas condiciones podemos rápidamente apreciar si las pruebas realizadas alcanzan a satisfacer el criterio establecido. La herramienta pierde fuerza con otros tipos de criterios que no se basan en la cantidad de código testeado (ej.: "un subconjunto Z del dominio de datos de entrada").

Facilitar el estudio del código de las reglas de negocio implementadas en la base de datos presenta innegables beneficios. Dentro de los mas importantes, figuran la disminución de los costos de mantenimiento, la disminución de los costos de testing, la mejora en la calidad del software provisto (a consecuencia de un mejor testing), la generación de una documentación sencilla y legible sobre el testing realizado (así como de la estructura de las reglas al momento de la realización del test), etc …

La herramienta adquiere mayor relevancia aún en los casos en los que se quiera encarar una reingeniería de las reglas del aplicativo o bien del aplicativo existente limitándose a la base de datos activa. En estos casos, podemos acelerar el estudio de lo existente gracias al resultado producido por el ADB_Map.

Podríamos preguntarnos también, para abordar la pregunta inicial desde otra perspectiva, si es bueno usar triggers dentro de una aplicación. Si bien al comienzo de esta sección afirmamos que el uso de los triggers dentro del mercado actual es bastante limitada, creemos firmemente que la disponibilidad técnica de los triggers es muy útil y es cuestión de tiempo y conocimiento para ver proliferar el uso de los triggers. Como consecuencia de esta creencia es que confiamos en la utilidad de la herramienta desarrollada.

Los motivos que nos hacen creer en la utilidad de los triggers son varios y variados. Entre los mas importantes se encuentra la simplicidad para resolver situaciones varias detalladas en la sección 2.4. A esta simplicidad hay que agregarle la portabilidad del trigger. Al ser un trigger un objeto mas de la base de datos, no debemos preocuparnos por la compatibilidad del código fuente con relación al sistema operativo o al hardware subyacente.

Sin embargo, Oracle aconseja "you should only use triggers when necessary" (Usted sólo debería usar triggers cuando sea indispensable). Luego siguiendo la misma línea, la corporación afirma: "The excessive use of triggers can result in complex interdependencies, wich may be difficult to maintain in a large application" (El uso excesivo de triggers puede

generar complejas inter relaciones posiblemente de costoso mantenimiento en una gran aplicación).

Esto revela que hemos generado una herramienta que ayuda a resolver algunos problemas reconocidos y es deseable profundizar la investigación en esta dirección.

- **¿Cuales son las ventajas y desventajas de la utilización de la metodología desarrollada?**

El trabajo con el ADB_Tester dejó un saldo positivo pese a tornarse luego de unas ejecuciones un tanto "pesado" el procedimiento para las necesidades reales.

Las dificultades encontradas durante su utilización en el caso real elegido para su experimentación se concentraron en el testeo de las modificaciones. En esta etapa, nos vemos obligados metodológicamente a ejecutar reiteradamente todas las herramientas desarrolladas por mas que la modificación haya sido sencilla. Esto no es necesariamente imprescindible pues si no se modifica alguna regla de forma tal que su estructura de control se vea modificada, no es necesario ejecutar nuevamente el ADB_Map, ni el ADB_Debugger (siempre y cuando no se haya ejecutado previamente el ADB_Cleaner).

Lo que vimos antes como una dificultad también representa una ganancia de sumo interés. La utilización de esta herramienta, entrega al usuario / responsable una medida concreta del test realizado. A esto se agrega la posibilidad de encontrar fácilmente errores de "lógica" que hacen intransitables ciertos caminos. Gracias al marcado sobre el DBGraph que modela la base de datos activa en estudio, rápidamente llegamos al sector dónde se origina el problema.

Lo que se ha realizado para solventar lo tedioso que resulta reiterar varias veces la ejecución de las diferentes herramientas desarrolladas es mejorar la performance de las mismas. Los tiempos de respuestas de las componentes del ADB_Tester son realmente despreciables.

Otra ventaja a destacar de la metodología propuesta es la certificación documentada de las pruebas realizadas. La certificación proviene de la correctitud de la herramienta y la documentación del grafo coloreado junto a los datos de entrada.

Una apreciación rápida sobre el DBGraph producido por el ADB_Map nos permitió obtener ventajas estratégicas en el testing a realizar sobre el caso real visto en la sección 5.1. La ventaja mas importante fue sin duda, la confirmación de la no existencia de relaciones entre los diferentes reglas (no figuraban en el DBGraph ejes inter-trigger).

Al no observarse relaciones inter – reglas (triggers que detonen a otros triggers) pudimos llevar a cabo un testing modular, desentendiéndonos de la realización de un test de integración final entre todas las reglas.

En consecuencia, pudimos dividir a la etapa de test de la base de datos activa en pequeños tests propios a cada regla. Dichos tests, que podemos denominar "tests de regla", pudieron realizarse en forma totalmente independiente unos de otros y, entonces, se pudieron realizar por diferentes personas, en diferentes momentos y con diferentes criterios asociados. Esto quiere decir que no todos alcanzaron necesariamente el mismo nivel de cubrimiento sino que nos hemos dado por satisfechos si en una regla cubríamos a todos los nodos, en otra regla si cubríamos a todos los ejes, etc ...

Esta división realizada representa una importante ventaja pues disminuye la complejidad del test a realizar.

Asimismo, en el ejemplo de la sección 5.2, dado el criterio de cubrimiento a alcanzar, pudimos simplificar el test a realizar ya que observamos claramente las relaciones existentes entre las diferentes reglas presentes.

Esta misma estrategia para diseñar el testing, puede tener su lado negativo como hemos visto en el caso de estudio desarrollado en la sección 5.3. En este último ejemplo, el hecho de estar trabajando con una base de datos distribuida, generó una situación "confusa" para el ADB_Tester y en consecuencia pudo colocar al responsable del testing en problemas.

La visualización del grafo obtenido tras la ejecución del ADB_Map representó en consecuencia una ventaja importante del uso del ADB_Tester.

- **¿Ha sido acertada la forma de visualizar los DBGraphs?**

La afirmación con la que finalizamos la pregunta anterior, nos deja afirmar que sí. La visualización de nodos y ejes en un gráfico representa una interfase mucho mas amigable, natural e intuitiva que sus solas especificaciones (el archivo grafo.dot).

Ahora bien, nos podemos preguntar si la elección de la herramienta GraphViz fue la acertada. Teniendo en cuenta que hemos tomado la versión de GraphViz que corre sobre Windows y que hemos desarrollado el ADB_Tester sobre Unix, aparece el interrogante sobre la indispensabilidad de realizar el cambio de plataforma. Este cambio obliga al usuario a migrar el contenido del archivo resultante a una plataforma Windows cada vez que queremos visualizar el estado de nuestro test.

Este problema (si lo podemos considerar como tal) tiene su escapatoria para aquel que considere engorroso el procedimiento anteriormente descrito:

tomar los fuentes de GraphViz y adaptarlos a la plataforma deseada
utilizar a los ejecutables disponibles, si se trata de una plataforma contemplada
desarrollar la herramienta de visualización que se desee

Por las características explicitadas en la sección 4.5, nosotros creemos que la elección de GraphViz resultó totalmente satisfactoria.

- **¿Límites del testeo dinámico a través del ADB_Tester?**

El trabajar sobre una base de datos Oracle nos impone ciertos límites que creemos convenientes de recalcar:

- Imposibilidad de chequear prácticamente la finalización de un test (manejo de los cursores limitado de Oracle, sección 2.4.3).
- Problemas con la mutua recursividad entre las reglas, pues no son permitidas.

- **¿Cómo agilizar a la etapa de testing de una BDA Oracle?**

Utilización del ADB_Tester.

- **¿Se han verificado los resultados teóricos de [Bal97] sobre los diferentes criterios de cubrimiento vistos?**

La pregunta así formulada puede generar confusión. No intentamos en ningún momento chequear la correctitud del trabajo mencionado, pero sí intentamos ver si las conclusiones obtenidas se mantenían con las leves modificaciones producidas por Oracle en el modelo teórico Starburst.

Hecha esta aclaración, afirmamos que nosotros vemos respetadas las propiedades expuestas en [Bal97].

6.2 Aplicaciones futuras

6.2.1 Sobre el ADB_Tester

- Industrializar al ADB_Tester
- Relacionar a los nodos con el código asociado para permitir un mejor análisis a posteriori y potenciar las conclusiones.
- Diseñar e incorporar facilidades para la iteración de las pruebas.
- Brindar un entorno propio al ADB_Tester que integre a todas sus componentes.

6.2.2 Al costado del ADB_Tester

- Implementar una herramienta similar a ADB_Tester apta para DB2, Illustra, Informix, ...
- Realizar un trabajo similar considerando el "significado" del código estudiado. Esto posibilitaría obtener un DBGraph más realista. El sentido de la aseveración anterior es tener en cuenta ejes que se generarían frente a la construcción de sentencias SQL en tiempo de ejecución.

7 Apéndices

Arquitectura elegida para el desarrollo de la herramienta ADB_Tester

Entorno:
Oracle 7.1
Unix varios

Desarrollos
Pro*C

Fundamentos prácticos indispensables para la construcción de la herramienta ADB_Tester

Diccionario de datos de la base de datos Oracle.
Lenguaje C – Pro*C
Dominio del PL/SQL
Fundamentos prácticos de Unix

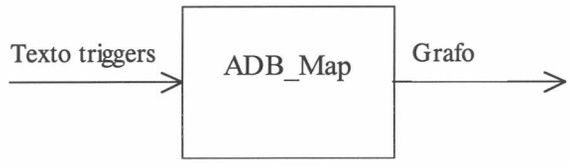
7.1 Documentación técnica del ADB_Map

7.1.1 Introducción

El objetivo del programa es obtener un grafo que represente la estructura interna (i.e. el mapa) de la aplicación a testear. Dicho grafo será el que se ha desarrollado en el cuerpo principal de esta tesis.

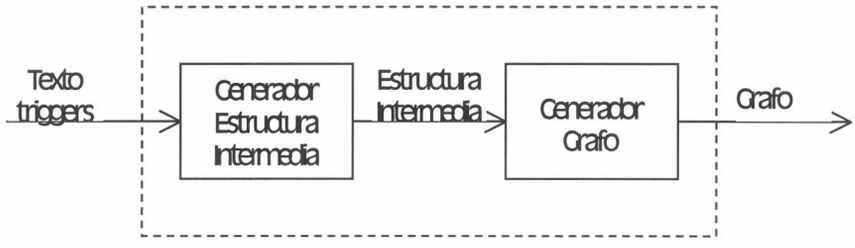
La aplicación a testear debe cumplir con las siguientes características:
Estar compuesta por un conjunto de triggers.
Los triggers se detonan entre sí sólo a través de sentencias SQL de insert, update o delete. No se tendrán en cuenta sentencias SQL preparadas dinámicamente pues no son detectables mediante un análisis sintáctico del código fuente de los triggers.
No se considerarán los efectos producidos por un "rollback" eventual.
No se permitirán sentencias que rompan la secuencialidad en el flujo de control (salvo el LOOP y el IF) como ser el tratamiento de una excepción.
Respetar las demás restricciones enumeradas mas adelante.

La entrada de ADB_Map es el conjunto de triggers que conforman la aplicación, la salida es el grafo que los representa. En esta implementación, la aplicación está compuesta por todos los triggers correspondientes a las tablas de un usuario dado como parámetro.



Para determinar el grado de cubrimiento que tiene la aplicación a testear, estudiamos el flujo de control de cada trigger. Esto permite saber que parte del trigger fue efectivamente ejecutada. Por ejemplo, si un programa tiene una estructura condicional *if then ... else ... end if*, es importante saber que parte del código fue probada (si fue el *then* o fue el *else*).

7.1.2 Procesos



7.1.3 Representación del Mapa de la Aplicación

Para ver con mayor desarrollo los puntos siguientes, remitirse al cuerpo principal de la tesis.

7.1.3.1 Estructuras de control

Las estructuras de control se representan a través de un grafo cuyos nodos representan conjuntos de instrucciones que se ejecutan secuencialmente en forma atómica y cuyos ejes representan la forma en que se puede encadenar la ejecución de los nodos.

7.1.3.2 Triggers

Un trigger se representa a través de la combinación de sus estructuras de control. Se llama nodo inicial o cabecera, al primer nodo de un trigger. Se llama nodo final o terminal al último nodo de un trigger. Todos los triggers tienen un solo nodo inicial y un solo nodo final; en el caso más simple en que no hay estructuras de control, coinciden el nodo inicial y el terminal. Se llama familia de nodos a todos los nodos que pertenecen a un mismo trigger.

7.1.3.3 Aplicación

Una aplicación se representa con todos los triggers que la componen más la representación de los eventos que los vinculan. Cualquier nodo de un trigger puede disparar el nodo inicial de otro trigger. El eje que los une representa la posibilidad sintáctica de detonación del trigger.

La aplicación se representa con un conjunto de nodos que pueden estar conectados o no, y n puntos de entrada diferentes, uno por cada trigger. Para obtener un DBGraph, se crea un nodo inicial único (denominado R000) y uno final único (denominado R999) de la aplicación, ficticios ambos. Del primero sale un eje hacia cada uno de los nodos iniciales de los triggers hallados en la base de datos. Este nodo es el único que no posee ningún eje de entrada. Todo nodo final de un trigger de la base de datos posee un eje que va hacia el nodo final de nuestro grafo, que no posee ejes emergentes.

7.1.3.4 Tipos de nodos

Nodo inicial: punto de entrada a un trigger. En este tipo de nodos se registra el "evento" que detona al trigger (en el código fuente de la herramienta se distingue a este tipo de nodo marcándolo como de tipo *trigger*).
Nodo común: representan los distintos conjuntos de instrucciones de un trigger surgidos de la estructura de control de un trigger (en el código fuente de la herramienta se considera a este tipo de nodo como de tipo *nodo*).
Nodo inicial y final de aplicación (el R000 y el R999 sin tipo asociado).

7.1.3.5 Tipos de ejes

Eje intra regla: representa el flujo causado por una estructura de control. Su nodo origen y destino pertenecen a la misma regla
Eje inter regla: representa el flujo causado por un evento detonado a través de un insert, update o delete. Su nodo origen pertenece a una regla y su nodo destino es el nodo inicial de otra regla.

7.1.4 Particularidades del diseño del ADB_Map

7.1.4.1 Funcionamiento general:

ADB_Map realiza los siguientes pasos para representar el mapa de toda la aplicación:

Representa cada uno de los triggers en forma aislada
Registra el evento que lo detonó.
Crea un grafo con los nodos necesarios para representar la estructura de control y los ejes intra regla.
Para cada nodo registra los eventos que detona.
Una vez representados los triggers en forma individual
Busca los eventos que detona cada uno de ellos
A partir de los eventos detonados crea los ejes inter reglas
Genera un archivo con los nodos y ejes, agregando los nodos ficticios de inicio y fin de aplicación (R000 y R999).

Como resultado final ADB_Map deja un archivo plano (llamado grafo.dot, en el directorio /tmp). Este archivo podrá ser utilizado por ADB_Tracer y por una herramienta de visualización como el GraphViz para estudiar el mapa de la aplicación.

7.1.4.2 Estructuras de datos principales:

t_evento: Evento que detona una regla o por el que una regla es detonada
accion: Acción (DELETE, INSERT, UPDATE) o combinación de acciones
tabla: Tabla sobre la que se ejecuta la acción
campos: Campos que se actualizan, separados por un espacio en blanco (solo en caso de UPDATE)

t_reglas: Estructura intermedia que contiene las reglas existentes
nombre: Nombre de la regla
tipo: Tipo de nodo: Trigger=> nodo cabecera de regla,
Código => nodo interno de regla
terminal: Indicador de ultimo nodo de una regla
cuando: Evento que la activa (solo para tipo trigger)
nsuc: Cantidad de sucesores
sucesor: Nombres de los nodos sucesores
ndetona: Cantidad de eventos que detona
detona: Eventos que detona
nllama: Cantidad de reglas invocadas explícitamente
llama: Reglas invocadas explícitamente

t_ejes: tabla de ejes
desde: Nodo origen
hasta: Nodo destino
nejes: Cantidad total de ejes

7.1.4.3 Fuentes que lo componen

M_grafo.pc: Punto de entrada a ADB_Map. Obtiene el código de cada uno de los triggers para que luego sea analizado.
Parse.c: Analiza la estructura interna y los eventos que detona un trigger y los representa en una estructura de datos.
Grafo.c: Crea los ejes inter reglas a partir de la representación de los triggers.
Evento.c: Contiene todas las funciones relativas a la estructura de evento, por ejemplo comparar, inicializar, etc.
Regla.c: Contiene todas las funciones relativas a la estructura de regla, por ejemplo agregar un sucesor, copiar, inicializar, etc.

7.1.4.4 Invocación

ADB_Map usuario password

7.1.4.5 Detalle de funciones de cada fuente

m_grafo.pc:

Se trata del punto de entrada del programa y contiene todo el acceso a la base de datos (Pro*C). Incluye las siguientes funciones:

- Main: Obtiene y verifica los parámetros de usuario y clave que recibe el programa. Inicializa la estructura de reglas. Invoca la función M_GrafoGenera para la construcción del grafo buscado. Genera dos archivos conteniendo el detalle de los nodos obtenidos (/tmp/grafo.dmp) y la especificación del grafo resultante (/tmp/grafo.dot). Dicha especificación coincide con la esperada por el graficador GraphViz.

- M_GrafoGenera: Contiene a toda la interacción de la herramienta con Oracle. Cualquier problema suscitado por esta interacción queda registrado en el archivo /tmp/ADB_Map.logerr.
  Obtiene de la base de datos cada uno de los triggers habilitados (status=enabled) y genera una regla por cada uno en la estructura P_ALL_REGLAS. La regla está compuesta por un nodo cabecera, n nodos internos y un nodo terminal. El nodo cabecera registra que evento provoca la detonación de la regla. Para resolver las dependencias internas del trigger invoca a la función ParseCode. Para facilitar el parsing, las sentencias se convierten en mayúsculas.
  La función devuelve 0 si terminó sin problemas y 1 en otro caso.

parse.c:

- ParseCode: Recibe un texto PL/SQL y el nodo a partir del cual tiene que cargar la estructura interna del texto. Detecta las diferentes estructuras de control que contiene el texto y las diferentes sentencias SQL que pueden disparar una regla. Termina el parsing cuando encuentra un delimitador de unidad de parsing.
  Las estructuras de control que reconoce son:
  IF ... THEN ... ([ELSEIF] ...) [ELSE] ... ENDIF
  WHILE ... DO ... END LOOP
  Donde: [ ] representa opcionalidad y ( ) representa más de una ocurrencia posible.
  Las sentencias SQL que reconoce son:
  INSERT
  DELETE
  UPDATE
  Cuando encuentra una sentencia SQL la agrega a la lista de eventos detonados por el nodo recibido como parámetro. Cuando encuentra una estructura de control genera los nodos intermedios y ejes correspondientes.
  Retorna un puntero al último nodo del texto parseado.

  ParseCode es una función recursiva. Se llama a sí misma para parsear el código dentro de las estructuras de control, por ejemplo entre un then y un else. Las unidades de parsing que puede recibir esta función son: un trigger completo, o el comprendido entre un then y un else, entre un then y un elsif, etc.

- ParseGetStatement: Dado un *string* busca la primer sentencia que encuentra. Para obtener la sentencia asume que:
  La sentencia incluye todo el texto hasta encontrar un ";"
  La sentencia comienza en el principio de *string* que recibe
  La sentencia que devuelve tiene:
  Todos los tabuladores y *new line* son transformados en espacios en blanco.
  Las secuencias de más de un espacio en blanco se reducen a uno solo

La sentencia no debe tener mas de 1000 caracteres

- ParseFillUpdateFields: Obtiene una secuencia de nombres de campos separados por blancos, a partir de un texto con la estructura campo=variable/valor, campo=variable/valor ... (texto de una sentencia de update entre SET y WHERE con la restricción de que no se permite preceder a los nombres de los campos por la tabla a la cual pertenecen).

- ParseFillEvent: Dada una sentencia SQL, completa la estructura de evento que la representa. Los eventos están compuestos por una acción (INSERT, UPDATE, o DELETE), por el nombre de una tabla (sobre la cual se efectúa la acción anterior), y en el caso de un UPDATE, por los campos de la tabla involucrados.

- StrNoBlanks: Dado un *string* devuelve el mismo *string* sin los espacios en blanco

regla.c:

- Reglainit: inicializa la cantidad de reglas existentes en cero.

- ReglaIsNull: chequea si una estructura de regla contiene datos (representa un nodo del grafo) o está vacía.

- ReglaNewCode: Agrega un nodo con los datos básicos (identificador y tipo del nodo) en la estructura de reglas, el resto de los atributos se inicializan como vacíos. Verifica que haya espacio en la estructura de reglas.

- ReglaNextName: dado el nombre de la regla sobre el cual se está trabajando, devuelve el identificador del nodo correspondiente concatenando al nombre del trigger el número de nodo corriente (P_REGLAS).

- ReglaPutSuc: pone al nodo pasado como primer parámetro como sucesor del nodo pasado como segundo parámetro.

- ReglaCopy: realiza duplicación exacta de nodos.

- ReglaAddDetona: Agrega un evento SQL (insert, update, delete y sobre que tabla y campos) a un nodo. Los nodos poseen los eventos que detonan asociados.

- ReglaEneDetona: devuelve el enésimo evento asociado al nodo.

- ReglaFindDetona: busca la existencia de un evento dado asociado a un nodo dado.

- ReglaGetNombre: devuelve el nombre de un nodo.

- ReglaEvent2Char: dado un evento devuelve un *string* cuyo texto describe en español el evento. Utilizada para el *debug* de la herramienta.

- Regla2Char: dado un nodo devuelve un *string* cuyo texto describe en español todos los atributos del nodo.

- ReglaDump: genera un archivo (/tmp/grafo.dmp) con el resultado de Regla2Char aplicado a los n (parámetro recibido) primeros nodos existentes en P_ALL_REGLAS.

grafo.c:

- GrafoEjes: devuelve en la estructura pasada por parámetro los ejes del grafo. No se incluyen los ejes que salen del eje inicial del grafo (el R000) ni los que llegan al nodo final del mismo (el R999). Primero se obtienen los ejes intra-triggers y luego los inter-triggers.

- GrafoDump: es la función encargada de generar el archivo resultante del proceso (/tmp/grafo.dot). Genera en primer lugar los ejes relacionados al R000, luego los ejes obtenidos mediante la función GrafoEjes, y por último, los ejes relacionados al R999.

evento.c:

- EventoAdd: dado el nombre de una tabla y una acción (insert, update, delete) completa el evento (t_evento) pasado como parámetro.

- EventoAddCampos: dado un *string* con los campos de una tabla sobre los que se realizará el update, completa el evento pasado como parámetro. Los campos están separados por un espacio en blanco.

- EventoNull: inicializa el evento con acción, tabla y campos nulos.

- EventoGetTabla: devuelve el nombre de la tabla sobre la que se lleva a cabo la acción del evento.

- EventoGetAccion: devuelve el nombre de la acción que se lleva a cabo.

- EventoGetCampos: devuelve los campos de la tabla (separados por espacios en blanco) sobre la que se lleva a cabo el *"update"* del evento.

- EventoFindCampo: dado un evento y un campo, devuelve 1 si el campo forma parte del evento y 0 si no.

- EventoGetCampoEne: dado un evento y un número de campo, devuelve el n-esimo campo del evento.

- EventoCompara: devuelve 1 si los dos eventos pasados como parámetro son iguales y 0 en otro caso.

error.c

- HandleError: función que genera un archivo (/tmp/reglas.trc) con un mensaje recibido como parámetro.

util.c:

- TablaSinUser: recibe un *string* del tipo [user.]tabla y retorna un puntero a tabla.

- StrToUpper: convierte a todas las letras de un *string* en mayúsculas. Modifica al *string* de entrada.

- NextLine: dado un puntero a char, modifica la posición del puntero a la línea siguiente (caracter siguiente al \n). En caso de tratarse de una única línea la función retorna 0.

- NextWord: devuelve en el segundo parámetro la siguiente palabra del string recibido como primer parámetro. No modifica el primer parámetro. Restricción: la palabra debe tener un largo inferior a los 80 caracteres. Para realizar esto se saltean a los espacios en blanco, tabulaciones y saltos a la línea del inicio del *string*. Se considera finalizada la palabra cuando se encuentra alguno de los siguientes caracteres: '\0', '\n', ';', '\t' o ' '.

- GetWord: la única diferencia que presenta con la función anterior es que modifica al string recibido como primer parámetro. El puntero se desplaza al final de la palabra devuelta.

- PostWord: la única diferencia que presenta con la función anterior es que saltea a los comentarios (ie en PLSQL los comentarios son las palabras que se encuentras entre - - y el fin de línea) que pueda encontrar.

- UtilGetField(n, IFS, destino, data): devuelve el campo enésimo de data en destino utilizando a IFS como separador de campos. Si no encuentra lo solicitado devuelve 0, sino el largo del campo solicitado.

7.1.4.6 Restricciones suplementarias

- Asumiremos que los bloques de código PL/SQL presentan las siguientes restricciones:
  1. No se aceptan sentencias resueltas en tiempo de ejecución.
  2. Existe un conjunto de palabras reservadas (INSERT, UPDATE, DELETE, IF, ...) que no podrán ser utilizadas en constantes (entre las que se encuentran los textos estáticos).
  3. La sintáxis de los update es la siguiente:
     update {*table*|*view*} [*alias*]
     set *column* = *expr* [, *column* = *expr*] ...
     [**where** *condition*]

     y no se permitirá la siguiente sintáxis:
     update {*table*|*view*} [*alias*]
     set (*column* [, *column*] ...) = *subquery*
     [**where** *condition*]
  4. No se procesan los procedimientos catalogados (con la consecuencia que pueden pasarse por alto sentencias SQL y verse modificado el mapa de la aplicación)
  5. Los valores de los campos para los updates son "old" y "new" y no pueden ser renombrados.

- Restricciones de los triggers:
  1. usa los triggers de un usuario únicamente.
  2. Cantidad máxima de eventos que detona una regla: 100
  3. Una regla esta asociada a un único evento.

- Se adoptó como entorno de trabajo Oracle 7.1

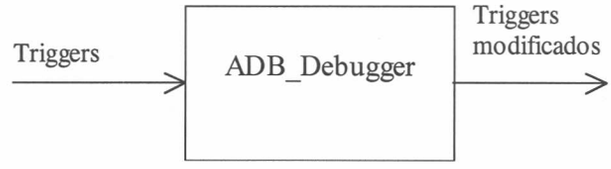
7.2 Documentación técnica del ADB_Debugger

7.2.1 Introducción

El objetivo del programa es modificar el texto de los triggers de modo tal que cuando se ejecuten dejen rastro de la secuencia recorrida, según las estructuras de control.

Arma el grafo que representa la estructura de control de cada trigger. Agrega una sentencia SQL en el inicio del texto de cada nodo. Cuando se ejecuta la sentencia inserta un registro en una tabla de log.

La entrada de ADB_Debugger es el conjunto de triggers que conforman la aplicación, la salida es el conjunto de triggers modificados. En este caso, la aplicación está compuesta por todos los triggers correspondientes a las tablas de un usuario.



7.2.2 Particularidades del diseño del ADB_Debugger

7.2.2.1 Funcionamiento general

ADB_Debugger realiza los siguientes pasos para lograr su objetivo:

- Forma un grafo representando las estructuras de control de cada trigger.
- Modifica el texto del trigger, agregando una sentencia SQL en el inicio del trigger y antes de cada estructura de control.
- Vuelve a crear el trigger en la base de datos con el texto modificado.

7.2.2.2 Estructuras de datos principales:

Las mismas que ADB_Map.

7.2.2.3 Fuentes que lo componen

M_ejec.pc: Punto de entrada a ADB_Debugger. Obtiene el código de cada uno de los triggers para que luego sea analizado.
Debug_log.pc: Funciones que agregan y quitan sentencias de log en el texto de un trigger.
Parsejec.c: Modifica el texto del trigger para que registre la secuencia de ejecución.
Evento.c: Descrito en ADB_Map.
Grafo.c: Descrito en ADB_Map.
Regla.c: Descrito en ADB_Map.

7.2.2.4 Invocación

ADB_Debugger usuario password

7.2.2.5 Detalle de funciones de cada fuente

m_ejec.pc:

- Main: Obtiene y verifica los parámetros de usuario y clave que recibe el programa. Inicializa la estructura de reglas. Invoca la función M_GrafoGenera para la construcción del grafo buscado.

- M_GrafoGenera: Contiene a toda la interacción de la herramienta con Oracle. Cualquier problema suscitado por ésta interacción queda registrado en el archivo /tmp/ADB_Map.logerr.
  Modifica el texto de los triggers, agregando sentencias SQL de forma tal que al ejecutarse dejen rastro del camino recorrido. Vuelve a crear los triggers

debug_log.c:

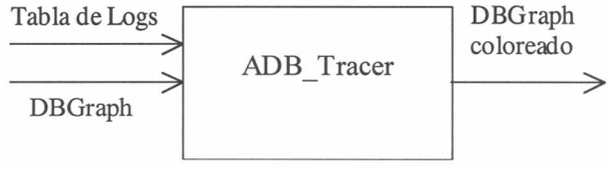
- InsertLog: dado el nombre de un trigger y su texto (o parte de él), agrega una primera sentencia (de log) al texto del trigger. Dicha sentencia genera log cuando se ejecuta el trigger.
- CleanLog: un texto elimina todas las sentencias de log con un formato específico (\nINSERT INTO SFX.ADB_TESTER_LOG (NODO, HORA) VALUES ...);
  Restricción: ADB_Debugger usa una tabla en el usuario sfx para generar debug.

7.3 Documentación técnica del ADB_Tracer

7.3.1 Introducción

El objetivo del programa es representar la ejecución de una aplicación a través del DBGraph obtenido con ADB_Map con sus caminos recorridos coloreados. La aplicación deja el rastro de los caminos efectivamente recorridos en una tabla de logs. El rastro se genera a través de sentencias SQL insertadas en la aplicación por el ADB_Debugger.

La entrada de ADB_Tracer es la tabla de logs y el DBGraph generado por el ADB_Map. La salida es el DBGraph modificado.



7.3.2 Particularidades del diseño del ADB_Tracer

7.3.2.1 Funcionamiento general

ADB_Tracer realiza los siguientes pasos para lograr su objetivo:

En caso que ocurra algún error durante la interacción con la base de datos, se registrará el mismo en el archivo /tmp/ADB_Tester.logerr.

7.3.2.2 Estructura principal

t_dot: Tabla que contiene los ejes y sus atributos para genera un archivo utilizable por GraphViz.

desde: origen de cada eje
hasta: destino de cada eje
atrib: atributos de cada eje
nejes: cantidad total de ejes

7.3.2.3 Fuentes que lo componen

m_intdot.pc: Punto de entrada a ADB_Tracer.
busca_log.pc: Recupera los logs que deja la aplicación al ser ejecutada.
marca.c: Modifica los atributos de los ejes de un DBGraph.
error.c: Descrito en ADB_Map.
util.c: Descrito en ADB_Map.

7.3.2.4 Invocación

ADB_Tracer usuario password

7.3.2.5 Detalle de funciones de cada fuente

m_intdot.pc:

- Main: Lee el DBGraph de una aplicación y cada uno de los nodos efectivamente ejecutados. A partir de los nodos arma el camino ejecutado y colorea los ejes del DBGraph. Como resultado obtiene un archivo legible por GraphViz.
- TomaDot: Transforma un archivo con las especificaciones de un DBGraph a una estructura para su procesamiento.
- GrabaDot: Transformación inversa a la realizada por la función anterior.

busca_log.pc:

- BuscaCorrida: En cada llamado devuelve un nodo efectivamente ejecutado, según el orden de ejecución.

marca.c:

- ExisteEje: Dados dos nodos verifica si existe un eje que los una.
- MarcaEje: Dados dos nodos cambia los atributos del eje cuyo nodo destino es el segundo parámetro y origen es el primero. Si es necesario marca adicionalmente los nodos ficticios inicial y final.

7.4 Documentación técnica del ADB_Clean

7.4.1 Introducción

El objetivo del programa es retirar del código fuente de los triggers las sentencias SQL insertadas por el ADB_Debugger.

La entrada de ADB_Cleaner es la tabla de logs y el DBGraph generado por el ADB_Map. La salida es el DBGraph modificado.

7.4.2 Particularidades del diseño del ADB_Cleaner

7.4.2.1 Funcionamiento general

ADB_Cleaner realiza los siguientes pasos para lograr su objetivo:

En caso que ocurra algún error durante la interacción con la base de datos, se registrará el mismo en el archivo /tmp/ADB_Cleaner.logerr.

7.4.2.2 Fuentes que lo componen

m_cleaner.pc: Punto de entrada a ADB_Cleaner.
debug_log.pc: Descrito en ADB_Debugger.

7.4.2.3 Invocación

ADB_Cleaner usuario password

7.4.2.4 Detalle de funciones de cada fuente

m_cleaner.pc:

- Main: Punto de entrada al programa.
- M_Cleaner: Dado un usuario, modifica todos los textos de los triggers para eliminar las sentencias de log agregadas por ADB_Debugger y vuelve a crear los triggers.

7.5 Misceláneas

7.5.1 Makefile utilizado

```
.SUFFIXES: .o .c .pc

#Variables y reglas para Pro*C
PROC16 = proc16
PCCINCLUDE = include=$(ORACLE_HOME)/sqllib/public
PCCFLAGS = $(PCCINCLUDE) host=c hold_cursor=no maxopencursors=100 release_cursor=yes
ltype=none ireclen=3000 oreclen=132 select_error=no sqlcheck=none lines=yes dbms=v6

# Oracle Compilation flags
LDORAFLAGS = -L$(ORACLE_HOME)/lib
ORACLE_LIBS = -lsql $(ORACLE_HOME)/lib/osntab.o -lsqlnet -lora -lsqlnet -lora -lnlsrtl -lcv6
-lcore -lnlsrtl -lcv6 -lcore -lm -lsocket

# C Flags
MYCFLAGS= -c -I$(ORACLE_HOME)/sqllib/public

all: cleaner map debugger tracer

clean:
	rm -rf *.o $(EXES)

# ADB Exes
EXES=ADB_Cleaner ADB_Map ADB_Debugger ADB_Tracer

# ADB Libs
LIB_SRCS=error.c util.c
LIB_PCSRCS=oralib.pc
LIB_OBJS=$(LIB_SRCS:.c=.o) $(LIB_PCSRCS:.pc=.o)

# ADB Cleaner
CLEANER_SRCS=
CLEANER_PCSRCS=m_cleaner.pc debug_log.pc
CLEANER_OBJS=$(CLEANER_SRCS:.c=.o) $(CLEANER_PCSRCS:.pc=.o)

cleaner: $(CLEANER_OBJS) $(LIB_OBJS)
	$(CC) $(CFLAGS) $(LDORAFLAGS) -o ADB_Cleaner $(CLEANER_OBJS) $(LIB_OBJS)
$(ORACLE_LIBS)

# ADB Map
MAP_SRCS=evento.c grafo.c parse.c regla.c
MAP_PCSRCS=m_grafo.pc
MAP_OBJS=$(MAP_SRCS:.c=.o) $(MAP_PCSRCS:.pc=.o)

map: $(MAP_OBJS) $(LIB_OBJS)
	$(CC) $(CFLAGS) $(LDORAFLAGS) -o ADB_Map $(MAP_OBJS) $(LIB_OBJS) $(ORACLE_LIBS)

# ADB Debugger
```

```
DEB_SRCS=evento.c grafo.c parsejec.c regla.c
DEB_PCSRCS=m_ejec.pc debug_log.pc
DEB_OBJS=$(DEB_SRCS:.c=.o) $(DEB_PCSRCS:.pc=.o)

debugger: $(DEB_OBJS) $(LIB_OBJS)
	$(CC) $(CFLAGS) $(LDORAFLAGS) -o ADB_Debugger $(DEB_OBJS) $(LIB_OBJS)
$(ORACLE_LIBS)


# ADB Trace
TRC_SRCS=m_intcbt.c marca.c
TRC_PCSRCS=busca_log.pc
TRC_OBJS=$(TRC_SRCS:.c=.o) $(TRC_PCSRCS:.pc=.o)

tracer: $(TRC_OBJS) $(LIB_OBJS)
	$(CC) $(CFLAGS) $(LDORAFLAGS) -o ADB_Tracer $(TRC_OBJS) $(LIB_OBJS) $(ORACLE_LIBS)

.pc.c:
	$(PROC16) $(PCCFLAGS) iname=$*.pc

.pc.o:
	$(PROC16) $(PCCFLAGS) iname=$*.pc
	$(CC) $(CFLAGS) $(MYCFLAGS) -c $*.c

.c.o:
	$(CC) $(CFLAGS) $(MYCFLAGS) -c $*.c
```

7.5.2 Otros fuentes que componen al ADB_Tester

```
debug_log.h
error.h
evento_fnc.h
grafo_fnc.h
grafo_typ.h
ora.h
oraca.h
oralib.pc
regla_fnc.h
sqlca.h
trace_typ.h
```

8 Bibliografía

[ABC82] Validation, Verification, and Testing of Computer Software (W. Richards Adrion, Martha a. Branstad, John C. Cherniavsky) Computer Surveys Vol. 14 nro 2, junio 1982

[Bal97] Martín Balzamo. "Testing sobre bases de datos activas", 1997.

[DHW94] U. Dayal, E. Hanson, and J.Widom. Active database systems. In Modern Database Systems: The Object Model, Interoperability and Beyond, p. 434 – 456. ACM Press, W. Kim, 1994

[FP97] Norman Fenton and Shari Lawrence Pfleeger, Software Metrics, A Rigorous & Practical Approach, 1997 (Second edition)

[Kin76] J. King: Symbolic execution and program testing, comm. ACM, 19, 7, Julio 1976

[Ora7a] Oracle7 Server - Concepts Manual

[Ora7b] Oracle7 Server – Application Developer's Guide

[RW82] Data Flow Analysis Techniques for Test Data Selection (Sandra Rapps and Elaine J. Weyuker) "IEEE Trans. Sotft. Eng. 1982" p.272-278

[WF90] Set-oriented production rules in relational database systems (J. Widom and S. Finkelstein). "ACM SIGMOD International Conference on Management of Data" (Atlantic City, New Jersey, 05/1990) p. 259-270

[Whi93a] General Overview of Software Testing (Lee J. White) "Scuola Estiva Software Testing Metodi e tecniche" 1993

[Whi93b] Integration testing (Lee J. White) p.79-90

[WO80] E. J. Weyuker and T. J. Ostrand, Theories of program testing and application of revealing subdomains, IEEE Transactions on Software Engineering, 6, 3, pages 236-246, May 1980