



UNIVERSIDAD NACIONAL DE COLOMBIA

Generación Semiautomática de Código PL/SQL a partir de Representaciones de Eventos Basadas en Esquemas Preconceptuales

Juan Sebastián Zapata Tamayo

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y de la Decisión

Medellín, Colombia

2019

Generación Semiautomática de Código PL/SQL a partir de Representaciones de Eventos Basadas en Esquemas Preconceptuales

Juan Sebastián Zapata Tamayo

Tesis presentada como requisito parcial para optar al título de:

Magister en Ingeniería de Sistemas

Director:

Ph.D. Carlos Mario Zapata Jaramillo

Línea de Investigación:

Ingeniería de Software

Grupo de Investigación:

Lenguajes Computacionales

Universidad Nacional de Colombia

Facultad de Minas, Departamento de Ciencias de la Computación y de la Decisión

Medellín, Colombia

2019

Dedicatoria

*Para mi papá y asesor, sin tí nada de esto
sería posible; gracias por guiarme en este camino.
A Kiki y a Andy por todo su amor y apoyo.*

Agradecimientos

Agradezco primeramente a la Universidad Nacional, en especial a la Facultad de Minas y al Área Curricular de Sistemas e Informática, por permitirme hacer parte del programa de becas que contribuyó en gran parte a que este sueño se hiciera realidad.

Mi mayor agradecimiento será para mi asesor y padre Carlos Mario Zapata Jaramillo, quien con su inmenso apoyo, sabiduría, paciencia, amor y motivación me alentaron a finalizar este camino académico.

Agradezco a Paola Noreña por todo su apoyo y colaboración en pro de la realización de esta Tesis de Maestría.

Finalmente, a los profesores Miguel David Rojas y Gloria Giraldo por su apoyo durante el tiempo académico.

Resumen

PL/SQL (extensión en lenguaje procedural al lenguaje estructurado de búsqueda, por sus siglas en inglés) es un lenguaje híbrido desarrollado en Oracle® para combinar las capacidades de los lenguajes de búsqueda con el enfoque procedimental para el desarrollo de aplicaciones. Algunos autores estudian la generación automática de código fuente a partir de modelos para mejorar el proceso de desarrollo de software. En algunas propuestas se genera código SQL a partir de diagramas como entidad relación y esquemas preconceptuales. En otras propuestas de ingeniería inversa se generan productos de trabajo como los grafos de flujo de datos y los modelos de arquitectura a partir de código SQL y PL/SQL. En relación con el código PL/SQL, en una propuesta se genera código PL/SQL a partir de OCL (lenguaje de restricción de objetos, por sus siglas en inglés). Finalmente, en algunas propuestas se representan gráficamente características relacionadas con elementos del código PL/SQL, como eventos, pero sin propuestas de traducción a ese lenguaje. Las propuestas mencionadas aún excluyen las posibles estructuras del lenguaje procedimental que se incluye en PL/SQL. Igualmente, las propuestas de ingeniería inversa tienen modelos poco claros para los interesados y una de las propuestas comienza la traducción desde OCL, un lenguaje adecuado para comunicación técnica sin la mediación de modelos. Por estas razones, en esta Tesis de Maestría se propone la generación semiautomática de código PL/SQL a partir de eventos basados en esquemas preconceptuales. Se espera mejorar los contenidos del código PL/SQL generado por medio de la definición de reglas para mantener la consistencia entre los modelos y el código. Esta propuesta se valida con un caso de estudio basado en esquemas que describen la formulación real de un proceso químico.

Palabras clave: Generación semiautomática de código, PL/SQL, consistencia, esquemas preconceptuales, eventos.

Abstract

PL/SQL (Procedure Language extension to Structured Query Language) is a hybrid language developed by Oracle® for combining the capabilities of query languages with the procedural approach for developing applications. Automated code generation from models has been a topic covered by some authors for improving the software development process. We can find proposals for generating SQL code from diagrams like entity-relationship and pre-conceptual schemas. We can also find reverse engineering proposals for generating work products like data flow graphs and architecture models from SQL and PL/SQL code. Related to PL/SQL code, we can find a proposal for generating PL/SQL code from OCL (Object Constraint Language). Finally, we can find some graphical representations of features related to elements of the PL/SQL code, *e.g.* events, but no translation is still proposed. The aforementioned proposals fail in generating all the possible structures of the procedure language provided by PL/SQL. In addition, reverse engineering proposals reveal unclear models for stakeholders and one proposal starts the translation from OCL, a language intended for technical communication with no mediation of models. For this reason, in this M.Sc. Thesis we propose the semi-automated generation of PL/SQL code from event representation based on pre-conceptual schemas. We aim to improve the contents of the generated PL/SQL code by defining rules intended to keep consistency among the models and the code. We validate our proposal with a case study based on schemas describing actual formulae from a chemical process.

Keywords: Semi-automated code generation, PL/SQL, consistency, pre-conceptual schemas, events

Contenido

	Pág.
Resumen	IX
Abstract	X
Contenido	XI
Lista de figuras	XIII
Lista de tablas	XV
Lista de Símbolos y abreviaturas	XVI
Introducción	1
1. Marco Conceptual de la problemática	4
1.1. Eventos.....	4
1.2. Esquemas preconceptuales.....	4
1.3. Eventos en esquemas preconceptuales.....	5
1.4. PL/SQL.....	7
2. Revisión de la literatura	9
2.1. Propuestas de generación de código SQL a partir de modelos y viceversa	9
2.2. Propuestas de generación de modelos a partir de código PL/SQL y viceversa.	11
2.3. Propuestas de generación automática de código desde esquemas preconceptuales	15
2.4. Propuestas de eventos en esquemas preconceptuales	17
2.5. Resumen de hallazgos.....	19
3. Planteamiento del problema	21
3.1. Formulación de la hipótesis	21
3.2. Preguntas de investigación.....	21
3.3. Formulación del problema	21
3.4. Justificación	22
3.5. Objetivos.....	23
3.5.1. General	23
3.5.2. Específicos	23
4. Propuesta de Solución	25
4.1. Definición de reglas heurísticas.....	25
4.2. Consideraciones especiales en relación con las reglas heurísticas	37

5. Validación de la propuesta de solución	39
5.1. Definición del caso de estudio	39
5.2. Aplicación paso a paso de las reglas heurísticas de transformación y funcionamiento del prototipo.....	41
6. Conclusiones y trabajo futuro	67
Conclusiones.....	67
Trabajo futuro.....	69
Referencias.....	71

Lista de figuras

	Pág.
Figura 2-1: Conversión del modelo entidad-relación en SQL.....	9
Figura 2-2: Conversión desde un esquema relacional a código SQL	10
Figura 2-3: Desde SQL hacia grafos de adyacencia	11
Figura 2-4: Ingeniería inversa de código SQL a HBase	12
Figura 2-5: Arquitectura para la generación del PIM a partir de código PL/SQL	12
Figura 2-6: Código PL/SQL y el diagrama de flujo de datos que explica su estructura...	13
Figura 2-7: Ejemplo de uso de la herramienta SQL-PL4OCL	14
Figura 2-8: Grafo de interacción de eventos	19
Figura 5-1: Esquema preconceptual del caso de estudio.....	40
Figura 5-2: Condiciones iniciales de las variables del caso de estudio.....	42
Figura 5-3: Estructura de la tabla VARIABLE	43
Figura 5-4: Valores asignados a la tabla VARIABLE	43
Figura 5-5: Condiciones iniciales de los parámetros del caso de estudio.....	44
Figura 5-6: Estructura de la tabla PARÁMETRO	45
Figura 5-7: Valores asignados a la tabla PARÁMETRO.....	45
Figura 5-8: Notación gráfica del evento TIEMPO PASA.....	46
Figura 5-9: Condicional inicial del evento TIEMPO PASA.....	46
Figura 5-10: Condicional anidado con el condicional inicial del evento TIEMPO PASA..	47
Figura 5-11: Restricción con asignación de variables del evento TIEMPO PASA.....	49
Figura 5-12: Notación completa del evento TIEMPO PASA.....	51
Figura 5-13: <i>Triggers</i> finalmente incorporados en la tabla VARIABLE	53
Figura 5-14: Representación gráfica del evento TIEMPO PASA	53
Figura 5-15: Condicional inicial del evento MEZCLA INICIA	53
Figura 5-16: Condicional anidado con el condicional inicial del evento MEZCLA INICIA	54
Figura 5-17: <i>Triggers</i> finalmente incorporados en la tabla VARIABLE	56
Figura 5-18: Notación completa del evento MEZCLA FINALIZA.....	57
Figura 5-19: Otros <i>triggers</i> finalmente incorporados en la tabla VARIABLE.....	58
Figura 5-20: Dependencias entre eventos del caso de estudio.....	59
Figura 5-21: Restricción asociada con la versión final de TIEMPO PASA.....	60
Figura 5-22: Restricción asociada con la versión final de MEZCLA INICIA.....	60
Figura 5-23: Condicional de disparo de CONCENTRACIÓN DE SUSTANCIA INCREMENTA	61

Figura 5-24: Relación dinámica atómica INSERTA de CONCENTRACIÓN DE SUSTANCIA INCREMENTA	62
Figura 5-25: Condicional de disparo de CONCENTRACIÓN DE SUSTANCIA INCREMENTA	65
Figura 5-26: Contenido de la tabla CONCENTRACIÓN DE SUSTANCIA una vez se ejecuta la relación dinámica y se activan los <i>triggers</i> del caso de estudio	66

Lista de tablas

	Pág.
Tabla 1-1: Símbolos de los esquemas preconceptuales	5
Tabla 1-2: Nuevos símbolos de los esquemas preconceptuales.....	6
Tabla 2-1: Reglas para la generación del diagrama entidad-relación y código SQL a partir de esquemas preconceptuales.....	16
Tabla 2-2: Reglas para la generación de código JSP y PHP desde esquemas preconceptuales.....	16
Tabla 2-3: Reglas para la generación de código JSP con SQL embebido para las relaciones dinámicas atómicas.....	17
Tabla 2-4: Caso de estudio para la generación de código y productos de trabajo desde esquemas preconceptuales.....	18
Tabla 2-5: Resumen de hallazgos para la generación de código PL/SQL desde esquemas preconceptuales.....	20
Tabla 4-1: Reglas heurísticas de transformación #1 y #2.....	26
Tabla 4-2: Reglas heurísticas de transformación #3 y #4.....	27
Tabla 4-3: Regla heurística de transformación #5	28
Tabla 4-4: Regla heurística de transformación #6	29
Tabla 4-5: Regla heurística de transformación #7	30
Tabla 4-6: Regla heurística de transformación #8	31
Tabla 4-7: Regla heurística de transformación #9	32
Tabla 4-8: Regla heurística de transformación #10	33
Tabla 4-9: Regla heurística de transformación #11	34
Tabla 4-10: Regla heurística de transformación #12	35
Tabla 4-11: Regla heurística de transformación #13	36

Lista de Símbolos y abreviaturas

Abreviaturas

Abreviatura	Término
<i>PL/SQL</i>	Procedure Language extension to Structured Query Language
<i>OCL</i>	Object Constraint Language
<i>SQL</i>	Structured Query Language
<i>JSP</i>	Java Server Pages
<i>PHP</i>	Personal Home Page
<i>DDL</i>	Data Definition Language
<i>DML</i>	Data Manipulation Language
<i>UN-Lencep</i>	Universidad Nacional de Colombia— Lenguaje para la especificación de esquemas preconceptuales

Introducción

Según Feuerstein (2012) en la corporación Oracle® se define un lenguaje que permite incrementar las capacidades de los lenguajes declarativos de búsqueda, específicamente el SQL (*structured query language*), con el fin de mejorar el desarrollo de aplicaciones incorporando capacidades procedimentales. El lenguaje PL/SQL (extensión en lenguaje procedural al lenguaje estructurado de búsqueda, por sus siglas en inglés) es una respuesta a esta necesidad específica y se caracteriza por una sintaxis que incorpora procedimientos automáticos y funciones, que posibilitan la auditoría a diferentes procesos de búsqueda, manejo y recuperación de información en bases de datos.

Una revisión sistemática de la literatura en relación con la generación de código PL/SQL a partir de modelos y viceversa permite recopilar diferentes propuestas que abordan esa generación de manera automática o semiautomática, en búsqueda de la automatización y el mejoramiento del proceso de desarrollo de aplicaciones centradas en las bases de datos. Teorey *et al.* (2005) generan código SQL a partir de diagramas entidad relación, en tanto que Chaverra (2011) y Zapata *et al.* (2011) realizan una labor similar, pero partiendo de esquemas preconceptuales, una propuesta que después Zapata *et al.* (2012) ejemplifican con un caso de estudio, y Chochlik *et al.* (2015) lo hacen mediante un esquema relacional gráfico. También existen propuestas de ingeniería inversa en las cuales se generan productos de trabajo como grafos dirigidos (Celko, 2011), el modelo relacional y los esquemas de HBase (Serrano *et al.*, 2015) para explicar el comportamiento del SQL. Methakullawat y Limpitakorn (2014) generan modelos de arquitecturas de diseño a partir de código PL/SQL y Habringer *et al.* (2014) proponen una herramienta para analizar código heredado con el fin de hacerlo más entendible para los interesados con grafos de flujo de datos. En relación con PL/SQL, Egea y Dania (2017) generan código PL/SQL a partir de sentencias en OCL (lenguaje de restricción de objetos, por sus siglas en inglés) y Behrend *et al.* (2009) exploran la relación entre los eventos temporales y los disparadores (*triggers*) de SQL, proponiendo una ampliación de este lenguaje. Zapata (2012) representa

gráficamente eventos que sirven para disparar procesos y eventos que hacen parte de los resultados de los procesos, características que se pueden relacionar con elementos del código PL/SQL. Finalmente, Zapata *et al.* (2014) generan un juego para explicar el funcionamiento de esas mismas estructuras.

Como se aprecia, existen diferentes propuestas que procuran algún tipo de conversión desde o hacia lenguajes como SQL o PL/SQL, empleando como punto de partida o de llegada diferentes esquemas conceptuales. Sin embargo, en esas propuestas aún se obvian las posibles estructuras del lenguaje procedimental que se incluye en PL/SQL y elementos como los eventos, que se pueden representar para los modelos de negocios. Además, algunas de las propuestas, relacionadas con ingeniería inversa, se basan en modelos de comunicación técnica, que se encuentran un poco lejos del lenguaje común de los interesados. Finalmente, una de las propuestas se emplea OCL como punto de partida para la traducción a PL/SQL, pero este lenguaje tiene una sintaxis más relacionada con los programadores y no se emplean modelos para mediar en la traducción.

Dadas las dificultades que se mencionan en relación con estas propuestas, en esta Tesis de Maestría se propone un conjunto de reglas heurísticas como mecanismo para la generación semiautomática de código PL/SQL, tomando como base eventos que se representan en los denominados esquemas preconceptuales. Se toman en consideración los diferentes elementos que hacen parte de los eventos, como las condiciones iniciales (variables y parámetros) y las diferentes operaciones automáticas que se realizan al interior de un evento, con el fin de generar sentencias PL/SQL basadas en disparadores (*triggers*), con la notación procedimental que los acompaña. Con estas reglas heurísticas se puede iniciar un proceso de traducción semiautomático que contribuye a la conservación de la consistencia entre los modelos (expresados en esquemas preconceptuales) y el código fuente de la aplicación. Para realizar la validación de las reglas, se define un caso de estudio basado en el esquema preconceptual que describe la fórmula para el cálculo de la cantidad de sustancia soluble en la concentración de una mezcla en un contenedor, que incluye dos tipos de sustancias: una líquida y otra soluble (Noreña *et al.*, 2019). Este proceso químico se adapta a condiciones reales y sirve para explicar la aplicación en PL/SQL que controla su comportamiento.

Esta Tesis de Maestría de estructura del siguiente modo: el Capítulo 1 constituye el marco conceptual de la problemática, donde se describen los eventos, los esquemas preconceptuales y el lenguaje PL/SQL; en el Capítulo 2 se expone la revisión de literatura, en la cual se describen propuestas para la generación desde y hacia lenguajes SQL y PL/SQL empleando varios esquemas conceptuales y otros lenguajes, además de describir la forma en que se estructuran los eventos en esquemas preconceptuales; en el Capítulo 3 se plantea el problema, incluyendo la hipótesis, las preguntas de investigación y los objetivos de esta Tesis de Maestría; en el Capítulo 4 se presenta la propuesta de solución, que incluye un conjunto de reglas heurísticas para la generación semiautomática de código PL/SQL a partir de esquemas preconceptuales; en el Capítulo 5 se realiza el caso de estudio que permite la validación de la propuesta y, finalmente, en el Capítulo 6 se discuten las conclusiones y el trabajo futuro que se derivan de esta Tesis de Maestría.

1. Marco Conceptual de la problemática



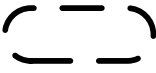


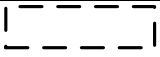
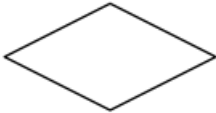

1.1. Eventos

Según Singh *et al.* (2009), los eventos se emplean en el ciclo de vida del desarrollo de software para indicar un suceso en el tiempo y lugar específico que inicia o finaliza un proceso que se desarrolle en el sistema, generando cambios en los estados y relaciones de este mismo. Weinbach y Garcia (2004) señalan los eventos como parte fundamental dentro de los requisitos funcionales y no funcionales, ya que estos proporcionan un entendimiento a fondo de los acontecimientos más importantes en los procesos de negocio que indican el inicio y el fin de una operación. Rosenzweig y Rakhimov (2009) establecen que en el mundo de las bases de datos existen eventos que pueden disparar la ejecución automática de un bloque de código, conocido como *trigger*. Ese tipo de eventos, conocidos como disparadores, se pueden asociar con modificaciones a los datos o a la estructura de la base de datos, a sucesos del sistema que alteran el encendido o el apagado de la base de datos o a otro tipo de eventos como el ingreso y salida de los usuarios del sistema.

1.2. Esquemas preconceptuales

Zapata *et al.* (2006) definen los esquemas preconceptuales como representaciones intermedias entre el lenguaje controlado y los esquemas conceptuales de UML. Como tales, los esquemas preconceptuales posibilitan la captura y representación del discurso de un interesado y su posterior traducción a los diferentes modelos que representan una aplicación de software. Zapata *et al.* (2011) establecen que estos esquemas se pueden emplear para la generación automática de código fuente. Los principales símbolos de los esquemas preconceptuales se presentan y discuten en la Tabla 1-1.

Tabla 1-1: Símbolos de los esquemas preconceptualesFuente: Zapata *et al.* (2006).

Símbolo	Discusión
 CONCEPTO	Sustantivos o sintagmas nominales que hacen parte del discurso de un interesado. Usualmente aluden a roles u objetos del mundo. Se puede distinguir entre conceptos clase (que preceden una relación “tiene”) o conceptos hoja (que reciben una relación “tiene” y de los cuales no parte ninguna relación.
 RELACIÓN ESTRUCTURAL	Verbos relacionados con características y relaciones permanentes entre los conceptos. Las únicas relaciones estructurales válidas del esquema preconceptual son los verbos “es” y “tiene”.
 RELACIÓN DINÁMICA	Verbos de operación o de acción, que normalmente aluden a procesos evidenciables en el discurso del interesado, tales como “registrar”, “calcular”, “autorizar”, etc.
 CONEXIONES	Enlaces que permiten conectar conceptos con relaciones estructurales/dinámicas o viceversa.
 REFERENCIAS	Elementos que permiten establecer la conectividad de una conexión cuando se usa para unir elementos distantes en el esquema.
 INSTANCIAS	Conjunto cerrado de posibles valores que se pueden asignar a un concepto hoja.
 CONDICIONAL	Aglutinador para expresar las condiciones que restringen la ejecución de una relación dinámica o un evento.
 IMPLICACIÓN	Relación causa-efecto entre relaciones dinámicas o entre condicionales y relaciones dinámicas. También se usa para expresar relaciones causa-efecto entre eventos.

1.3. Eventos en esquemas preconceptuales

Zapata *et al.* (2006) definen un elemento denominado “condicional”, que sirve para representar las precondiciones que se deben cumplir antes de la ejecución de un proceso y que, en consecuencia, podrían “disparar” ese proceso. Los condicionales se ligan con estos procesos (representados con las denominadas “relaciones dinámicas”) por medio de

implicaciones, que son causa-efecto entre el condicional y la relación dinámica. Posteriormente, Zapata (2012) establece un nuevo elemento que permite la representación de los eventos y que se denomina “relación eventual”. Este nuevo elemento, que se suele ligar con conceptos para generar los eventos, se refiere a verbos con valencia uno y que no requieren un agente (es decir, su complemento directo es un objeto) y tienen fuerza de disparadores. Tanto los eventos como las relaciones dinámicas se pueden especificar usando un símbolo similar al de los posibles valores y un lenguaje gráfico en forma de árbol. Noreña *et al.* (2019) definen un conjunto de nuevos elementos para los esquemas preconceptuales que permiten la especificación de los eventos. Esos elementos se listan y discuten en la Tabla 1-2.

Tabla 1-2: Nuevos símbolos de los esquemas preconceptuales

Parte 1/2. Fuente: Noreña *et al.* (2019)

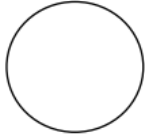
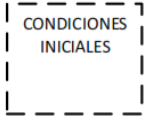
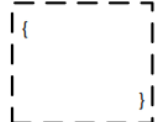


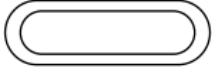




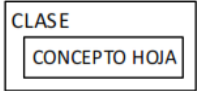
Símbolo	Discusión
 EVENTO	Conjuntos de conceptos/variables/parámetros con relaciones eventuales, que se usan como disparadores de los procesos (que usualmente se expresan mediante relaciones dinámicas). También, se pueden originar como resultado de una relación dinámica. En esta Tesis de Maestría sólo se consideran los eventos disparadores.
 CONDICIÓN INICIAL	Conjunto de valores de variables y parámetros que se asignan antes de iniciar la simulación del sistema mediante los esquemas preconceptuales.
 RESTRICCIÓN	Secuencia de operaciones, comparaciones y asignaciones que restringen el valor de un concepto, relación dinámica, evento o especificación.
 ESPEFIFICACIÓN	Conjunto de operaciones, condicionales, comparaciones y asignaciones que sirven para delimitar el alcance de un evento o relación dinámica.

Tabla 1-2: Nuevos símbolos de los esquemas preconceptuales

Parte 2/2. Fuente: Noreña *et al.* (2019)

Símbolo	Discusión
 JOINT	Elemento que sirve para unir implicaciones cuando se deban ejecutar conjuntamente para anteceder una relación dinámica o un evento.
 EVENTUAL	Tipo de relación que incluye verbos que usualmente no requieren un agente que los ejecute. Por ejemplo: “llega”, “aparece”, “inicia”, etc.
 VARIABLE	Concepto que no se liga con ningún concepto clase y cuyo valor puede variar continuamente durante la simulación del sistema que se expresa en el esquema preconceptual.
 PARÁMETRO	Valor que usualmente permanece constante durante toda la simulación del sistema que se expresa en el esquema preconceptual.
 OPERADOR	Símbolo que incluye los tipos de operaciones que se permiten en el esquema preconceptual como aritméticos (+, *, -, /), trigonométricos (seno, coseno, tangente, etc.), comparativos (máx, min, >, =, etc.) y propias de la simulación. Con línea punteada significa comparación y con línea continua significa asignación.
 OPERADOR	Enlace que sirve para unir operadores con conceptos y posibles valores para conformar un árbol.
 CONCEPTO-CLASE	Nodo que permite evaluar un conjunto concepto_clase/concepto_hoja.

1.4. PL/SQL

Feuerstein (2012) define PL/SQL como un lenguaje de programación y lenguaje de extensión de procedimientos para SQL que desarrollaron en Oracle® en 1988. El lenguaje funciona como conexión entre tecnologías de bases de datos y aplicaciones de procedimiento, de forma que se combinan armónicamente sentencias del lenguaje SQL (*Structured Query Language*) y sentencias correspondientes a lenguajes procedimentales.

De esta manera, el lenguaje PL/SQL comparte las características de ambos formalismos, combinando sentencias declarativas con procedimentales.

Uno de los elementos fundamentales del PL/SQL para efectos de esta Tesis de Maestría es el *trigger*, cuya sintaxis es la siguiente:

```
CREATE OR REPLACE TRIGGER NombreTrigger
{BEFORE|AFTER|INSTEAD OF}
{DELETE|INSERT|UPDATE [OF NombreColumna
[, NombreColumna...]]}...
ON {NombreTabla|Nombre-Vista}
[[REFERENCING {OLD [AS] NombreViejo | NEW [AS] NombreNuevo}... ]
FOR EACH {ROW|STATEMENT} [WHEN (Condición)]]
Bloque PL/SQL
```

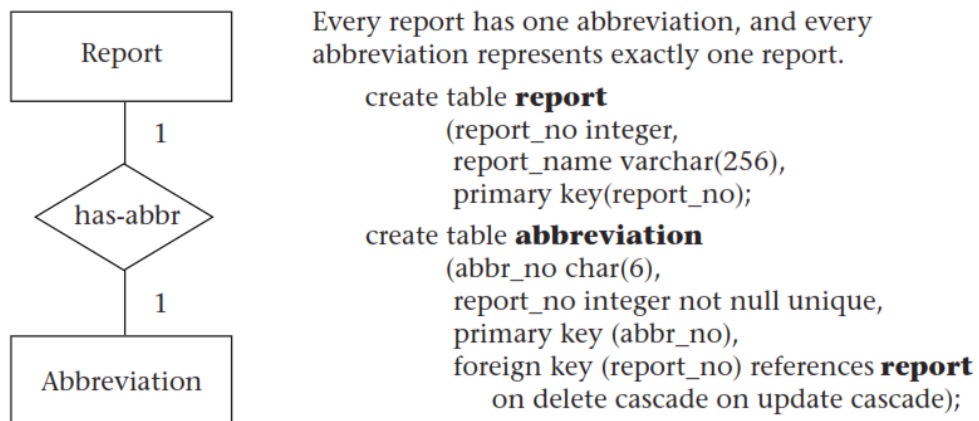

2.Revisión de la literatura

2.1. Propuestas de generación de código SQL a partir de modelos y viceversa

Teorey *et al.* (2005) establecen un conjunto de reglas de transformación a código SQL tomando como base el modelo entidad-relación. En la Figura 2-1 se puede apreciar el tipo de conversión que se realiza, en la cual la información aparece incompleta en el modelo, pero mucho más completa en el código. Si bien no es claro de dónde se extrae toda la información necesaria para crear las sentencias SQL, este antecedente constituye una prueba de que un esquema similar al preconceptual (en este caso el modelo entidad-relación) tiene la posibilidad de uso para la generación de sentencias declarativas de SQL. Además, el código generado no se relaciona con los elementos de PL/SQL, tales como los eventos disparadores, que permiten la ejecución de *triggers*.

Figura 2-1: Conversión del modelo entidad-relación en SQL

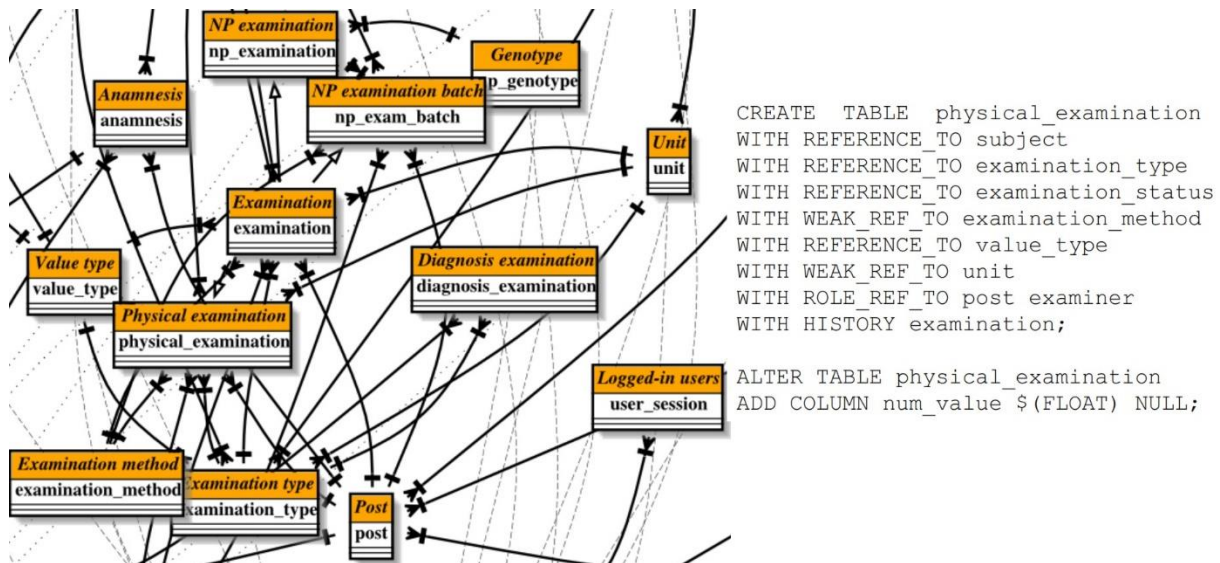
Fuente: Teorey *et al.* (2005)



Chochlik *et al.* (2015) realizan algo similar desde un esquema relacional gráfico que emplean para recuperar la información concerniente a las cláusulas SQL que desean generar (véase la Figura 2-2). El esquema no es claro en cuanto a su contenido, pues el código generado posee más información que el esquema del cual se genera, por lo cual las reglas para la generación de las sentencias SQL no se pueden reproducir. Además, se trata de código SQL y sus elementos propios, en los cuales los elementos procedimentales del PL/SQL no se pueden generar.

Figura 2-2: Conversión desde un esquema relacional a código SQL

Fuente: Adaptación de Chochlik *et al.* (2015)



Celko (2011) emplea los denominados grafos de adyacencia para comprender sentencias SQL en un proceso de ingeniería inversa que se ejemplifica en la Figura 2-3. En este caso, sólo una parte de la información de las sentencias SQL (los *insert*) se puede visualizar en el grafo, dejando poca claridad en cuanto a la conversión de la estructura. Además, se trata de ingeniería inversa (un proceso inverso al que se plantea en esta Tesis de Maestría) sobre sentencias estructurales, que poco se relacionan con el comportamiento que denotan los eventos en sentencias PL/SQL.

Figura 2-3: Desde SQL hacia grafos de adyacencia

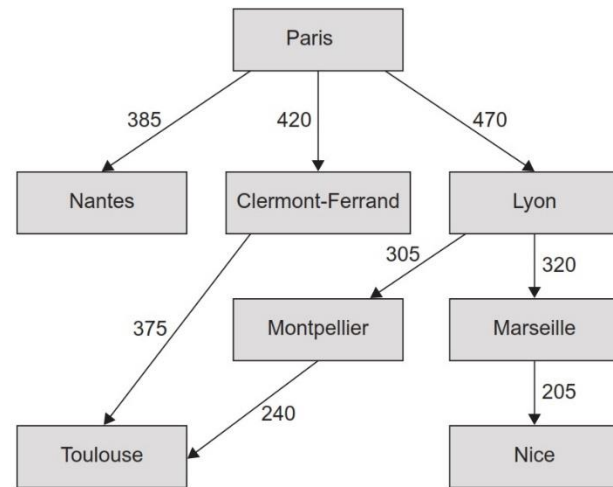
Fuente: Adaptación de Celko (2011)

```

CREATE TABLE Journeys
(depart_town VARCHAR(32) NOT NULL,
arrival_town VARCHAR(32) NOT NULL,
CHECK (depart_town <> arrival_town),
PRIMARY KEY (depart_town, arrival_town),
jny_distance INTEGER NOT NULL
CHECK (jny_distance > 0));

INSERT INTO Journeys
VALUES ('Paris', 'Nantes', 385),
('Paris', 'Clermont-Ferrand', 420),
('Paris', 'Lyon', 470),
('Clermont-Ferrand', 'Montpellier', 335),
('Clermont-Ferrand', 'Toulouse', 375),
('Lyon', 'Montpellier', 305);

```



Como se muestra en la Figura 2-4, Serrano *et al.* (2015) también realizan ingeniería inversa desde sentencias SQL pero esta vez para generar un modelo relacional intermedio que posibilita la transformación definitiva al lenguaje gráfico HBase, que se emplea en aplicaciones de bases de datos espaciales. Las sentencias de partida son de manipulación de datos y con ellas se generan vistas en HBase. El proceso es, nuevamente, contrario a lo que se pretende en esta Tesis de Maestría y los eventos no tienen forma de representación en este proyecto.

2.2. Propuestas de generación de modelos a partir de código PL/SQL y viceversa

Methakullawat y Limpiyakorn (2014) proponen una arquitectura (véase la Figura 2-5) para la generación automática de modelos independientes de la plataforma (PIM, por sus siglas en inglés) tomando como base código legado PL/SQL, con el fin de comprender su estructura y facilitar su actualización hacia otros lenguajes. En dicha arquitectura, a partir de las herramientas Oracle® se genera un código XML con el PL/SQL embebido que se lleva paulatinamente a un diagrama de clases que almacena su estructura definitiva. Si bien el punto de partida es código PL/SQL, el diagrama de clases resultante sólo es un

reflejo de la estructura del código, mas no refleja su contenido y, en consecuencia, los *triggers* se terminan almacenando en un atributo del diagrama de clases sin definir gráficamente su lógica de negocio.

Figura 2-4: Ingeniería inversa de código SQL a HBase

Fuente: Adaptación de Serrano *et al.* (2015)

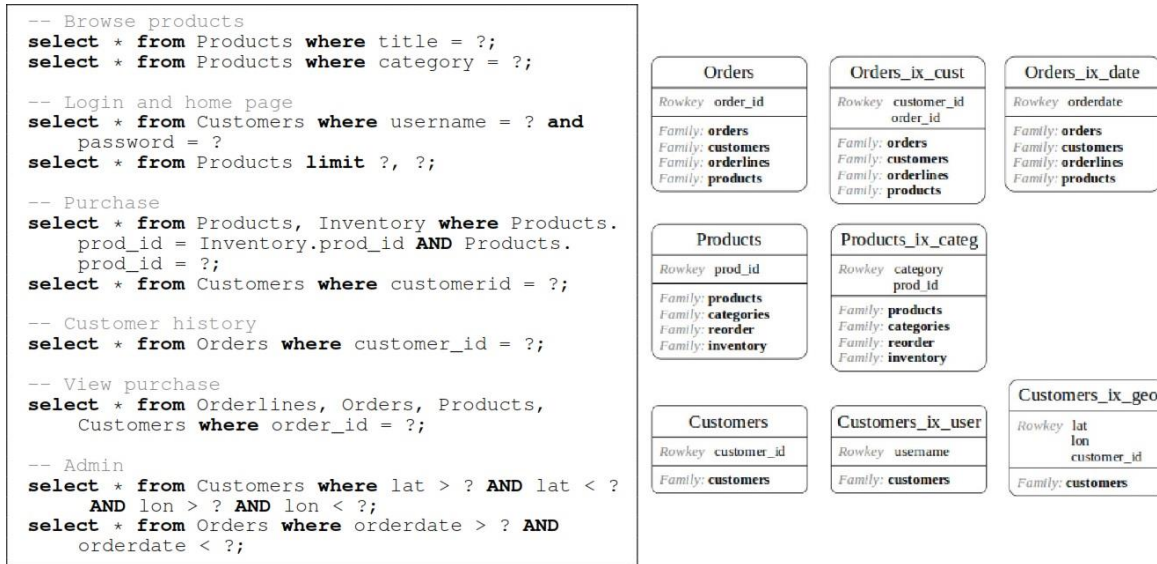
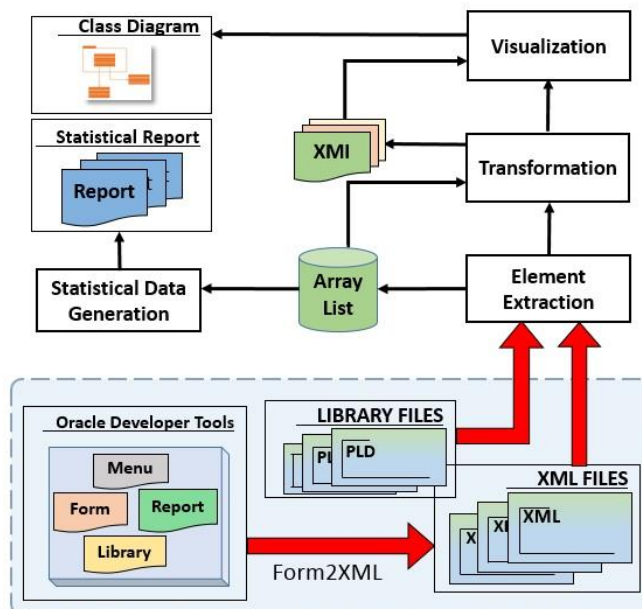


Figura 2-5: Arquitectura para la generación del PIM a partir de código PL/SQL

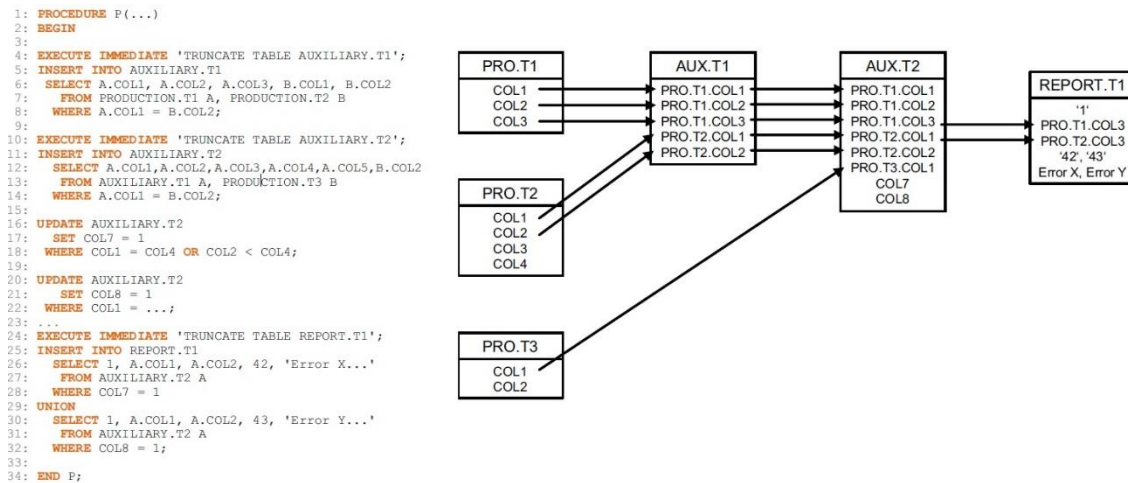
Fuente: Methakullawat y Limpiyakorn (2014)



De forma similar, Habringer *et al.* (2014) realizan un proceso de ingeniería inversa similar, al proponer una herramienta que analiza el código heredado PL/SQL y lo transforma en un diagrama de flujo de datos que refleja la estructura del código, como se muestra en la Figura 2-6. Nuevamente, el modelo sirve para explicar la estructura del código PL/SQL, pero no refleja la lógica del negocio que motiva el código.

Figura 2-6: Código PL/SQL y el diagrama de flujo de datos que explica su estructura

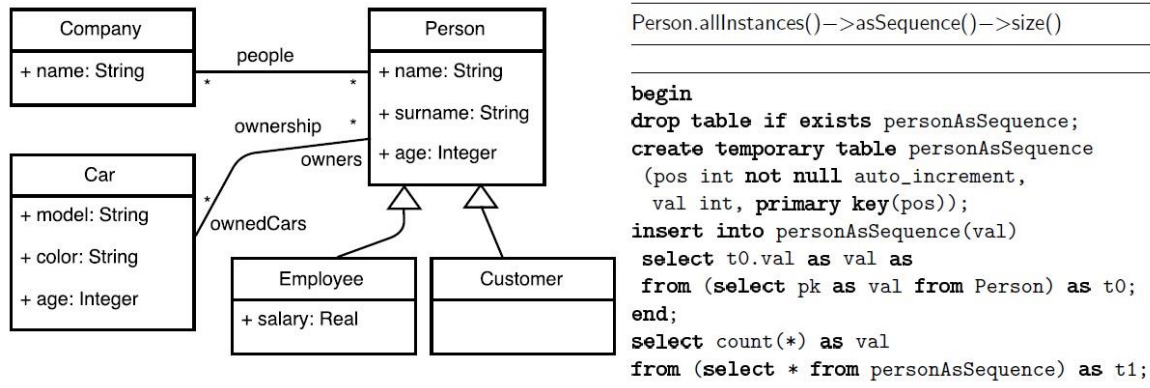
Fuente: Adaptación de Habringer *et al.*(2014)



Egea y Dania (2017) proponen la herramienta SQL-PL4OCL, que requiere un modelo de entrada y una expresión en lenguaje OCL (*Object Constraint Language*) para generar expresiones que combinan la parte declarativa y procedimental del código PL/SQL. A modo de ejemplo, en la Figura 2-7 se muestra el diagrama de clases de una compañía y una expresión en OCL que permite el cálculo del tamaño de la tabla cliente. El código PL/SQL de la figura se genera completamente con estos dos insumos. En este caso, el modelo de partida refleja las características del negocio, pero se requiere como entrada un insumo que sólo puede proporcionar personal técnico, como es el conjunto de expresiones en OCL. Sin embargo, esta herramienta constituye un avance en la generación automática de código PL/SQL a partir de modelos, ya que se generan las expresiones declarativas del SQL junto con las sentencias procedimentales del PL/SQL. Si bien sólo se genera código de procedimientos almacenados, esta herramienta constituye un punto de partida para la generación de eventos disparadores para bloques de *triggers*.

Figura 2-7: Ejemplo de uso de la herramienta SQL-PL4OCL

Fuente: Adaptado de Egea y Dania (2017)



Behrend *et al.* (2009) reconocen las limitaciones de las expresiones PL/SQL, debido a que sólo se pueden ejecutar por la interacción con la base de datos o mediante la llegada de un evento específico relacionado con variables del sistema, como la activación de la base de datos o el ingreso de un usuario, y proponen una extensión a la sintaxis de este lenguaje para incluir eventos temporales. Así, proponen las expresiones AT, EVERY y FROM como expresiones que disparan la ejecución de un bloque de *trigger* cuando se cumplen condiciones temporales. En el código siguiente se resalta la propuesta de uso de estas expresiones.

```

CREATE TRIGGER t4
FROM 2009-05-23 12:15 EVERY 7 DAYS
WHEN (SELECT count(*) FROM conf_credits)>0
BEGIN ATOMIC
UPDATE studs_rec
SET time=systimestamp, credit=credit+(SELECT conf_credits.credit
FROM conf_credits WHERE studs_rec.name= conf_credits.name);
END

```

Si bien el punto de partida es también código PL/SQL, es importante señalar que el manejo del tiempo y las expresiones temporales se constituye en una necesidad cuando se trata

de obtener código PL/SQL a partir de modelos. Esta propuesta tiene como principal inconveniente el hecho de que las sentencias que se proponen para el manejo temporal no corresponden a la sintaxis estándar del PL/SQL y, por ello, no se reconocen en los editores convencionales de este lenguaje.

2.3. Propuestas de generación automática de código desde esquemas preconceptuales

Chaverra (2011) y Zapata *et al.* (2011) reconocen la necesidad de generación de código SQL a partir de modelos más cercanos al dominio del interesado y por eso emplean como punto de partida los esquemas preconceptuales (Zapata *et al.*, 2006). De esta manera, proponen una serie de reglas heurísticas que también incluyen la generación de productos de trabajo intermedios como el diagrama entidad relación (véase la Tabla 2-1) y elementos de una aplicación final, como el código del controlador (en este caso JSP o *Java Server Pages*) y código de visualización (PHP o *Personal Home Page*), como se muestra en la Tabla 2-2. Además, Chaverra (2011) y Zapata *et al.* (2011) proponen una sintaxis que posibilita la especificación de las relaciones dinámicas, de forma tal que se puede generar el código de los métodos en un lenguaje de programación, incluyendo el código declarativo SQL para hacer el manejo de los datos. En la Tabla 2-3 se muestran las equivalencias en código JSP con instrucciones SQL embebidas para las relaciones dinámicas atómicas, que constituyen los elementos básicos sobre los cuales se construyen los métodos en el lenguaje de programación a partir de las relaciones dinámicas, pues se trata de combinaciones de operaciones simples para expresar la semántica de cualquier operación en una base de datos.

Si bien la propuesta de Chaverra (2011) y Zapata *et al.* (2011) es supremamente completa para la generación de código en SQL, JSP y PHP para la construcción de una aplicación completa, no se toma en cuenta una representación de los eventos en esquemas preconceptuales, que constituyen la base para la construcción de sentencias SQL asociadas con los *triggers*. Finalmente, en la Tabla 2-4 se muestra el código que se genera con un caso de estudio que realizan Zapata *et al.* (2012) para demostrar la efectividad de las reglas heurísticas que posibilitan la creación de una aplicación funcional.

Tabla 2-1: Reglas para la generación del diagrama entidad-relación y código SQL a partir de esquemas preconceptuales

Fuente: Chaverra (2011)

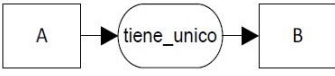
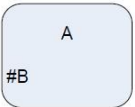
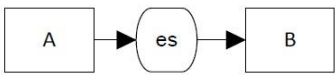
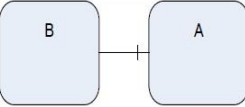
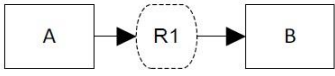
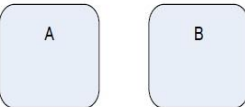
EP	ENTIDAD-RELACIÓN	SQL
		<pre>CREATE TABLE A (B varchar(30) default NULL, UNIQUE (B))</pre>
		<pre>CREATE TABLE B () CREATE TABLE A UNDER B ()</pre>
		<pre>CREATE TABLE A () CREATE TABLE B ()</pre>

Tabla 2-2: Reglas para la generación de código JSP y PHP desde esquemas preconceptuales

Fuente: Chaverra (2011)

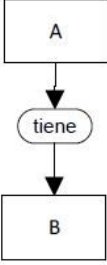

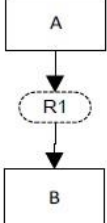
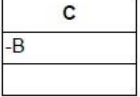
EP	Precondición	JSP	PHP
		<pre>public class A { private B b; } public class B { }</pre>	<pre><?php class A { var \$b = new B(); } class B { } ?></pre>
		<pre>public class C { private TIPO b; } public TIPO R1(TIPO b){ } }</pre>	<pre><?php class C { var \$b; } function R1(\$b){ } ?></pre>

Tabla 2-3: Reglas para la generación de código JSP con SQL embebido para las relaciones dinámicas atómicas

Fuente: Chaverra (2011)

Función	JSP
Lista	<pre>public List lista(condicion) { List consulta = this.query('SELECT * FROM MODELO WHERE condicion'); return consulta; }</pre>
Busca	<pre>public List buscar(id) { consulta = this.query('SELECT * FROM MODELO WHERE id=id'); return consulta; }</pre>
Selecciona	<pre>public List selecciona(campo = "nombre") { consulta = this.query('SELECT id, \$campo FROM MODELO'); return consulta; }</pre>
Inserta	<pre>function inserta(atributo1, atributo2...){ String sql = "INSERT INTO MODELO (atributo1, atributo2...) VALUES (atributo1, atributo2....)"; consulta = this.query(sql); }</pre>
Edita	<pre>function edita(id, atributo1, atributo2...){ sql="UPDATE MODELO SET atributo1= 'atributo1', atributo2= 'atributo2' ... WHERE id=id"; consulta = this.query(sql); }</pre>
Elimina	<pre>function elimina(id){ sql = "DELETE FROM MODELO WHERE id= 'id' "; consulta = this.query(sql); }</pre>

2.4. Propuestas de eventos en esquemas preconceptuales

Zapata (2012) reconoce las carencias de los esquemas preconceptuales para la representación de eventos define un nuevo símbolo para los eventos, que se basa en la presencia de las denominadas relaciones eventuales (verbos que no requieren la intervención de un agente, como llegar, surgir, iniciar, etc.) acompañadas de conceptos o variables del dominio.

Tabla 2-4: Caso de estudio para la generación de código y productos de trabajo desde esquemas preconceptuales

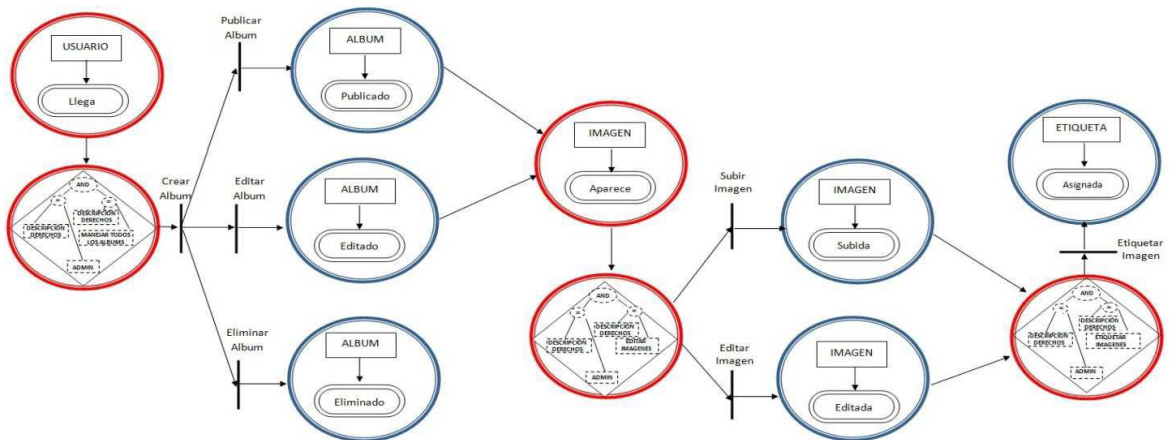
Fuente: Zapata *et al.* (2012)

EP	ER	SQL	MODELO
		<pre>CREATE TABLE administradores (id int(11) NOT NULL auto_increment, nombre varchar(255) default NULL, apellido varchar(255) default NULL, identificacion varchar(255) default NULL, PRIMARY KEY (id), UNIQUE KEY identificacion (identificacion))</pre>	<pre>class Administrador{ var nombre; var apellido; var identificacion;} </pre>
		<pre>CREATE TABLE grupos (id int(11) NOT NULL auto_ increment, horario varchar(255) default NULL, aula varchar(255) default NULL, codigo varchar(255) default NULL, asignatura_id int(11) default NULL, profesor_id int(11) default NULL, PRIMARY KEY (id), UNIQUE KEY codigo (codigo))</pre>	<pre>class Grupo{ var horario; var aula; var codigo; var asignatura = new Asignatura(); var profesor = new Profesor();} </pre>

Noreña (2013) utiliza esa definición para establecer un mecanismo de consistencia entre los diferentes productos de trabajo que hablan sobre eventos, como el diálogo controlado, las tarjetas de educación, el diagrama de procesos, las máquinas de estados y un nuevo diagrama que ella propone denominado grafo de interacción de eventos y que se puede apreciar en la Figura 2-8. Noreña *et al.* (2019) complementan esa definición con la incorporación de otros elementos para especificar los eventos y que se pueden apreciar en la Tabla 1-2. Finalmente, Zapata *et al.* (2014) validan con un juego el uso de la representación de eventos con esquemas preconceptuales empleando para ello el grafo de interacción de eventos.

Figura 2-8: Grafo de interacción de eventos

Fuente: Noreña (2013).



La incorporación de una sintaxis para representar los eventos en el esquema preconceptual permite la incorporación de estos elementos en el análisis del comportamiento de las aplicaciones de software desde el modelado. La consistencia de estos elementos se analiza sobre diferentes productos de trabajo intermedios, pero no se presenta en estas propuestas una traducción hacia el código fuente ni se proponen las posibles equivalencias que se requerirían para su incorporación en el desarrollo de la aplicación

2.5. Resumen de hallazgos

En cuanto al lenguaje SQL, es posible encontrar trabajos en la literatura que realizan la generación de sentencias a partir de modelos y viceversa, pero estos trabajos no incluyen las características procedimentales que sí son comunes al PL/SQL como extensión de este lenguaje. También, es común encontrar herramientas que realizan ingeniería inversa sobre sentencias PL/SQL hacia diferentes tipos de esquemas conceptuales, lo cual constituye un proceso inverso al que se plantea en esta Tesis de Maestría y que genera esquemas que no son claros para los interesados. Hay un trabajo que plantea la conversión del lenguaje OCL en sentencias PL/SQL, pero en este trabajo no hay mediación de modelos que permitan la comunicación con el interesado. Finalmente, se pueden encontrar en la

literatura trabajos que permiten la generación de código procedimental o declarativo desde esquemas preconceptuales, pero no simultáneamente a código tipo PL/SQL. Además, aparecen nuevas características en los esquemas preconceptuales que pueden posibilitar la transición hacia PL/SQL, dado el parecido que tienen los eventos en esquemas preconceptuales con las sentencias PL/SQL. En la Tabla 2-5 se sintetizan estos hallazgos, que dan origen a esta Tesis de Maestría.

Tabla 2-5: Resumen de hallazgos para la generación de código PL/SQL desde esquemas preconceptuales

Fuente: Elaboración propia

Autor(es)	Lenguaje/modelo inicial	Lenguaje/modelo final	Nivel de automatización	Manejo de eventos
Teorey <i>et al.</i> (2005)	Entidad-relación	SQL	Manual	No
Chochlik <i>et al.</i> (2015)	Esquema relacional	SQL	Manual	No
Celko (2011)	SQL	Grafos de adyacencia	Manual	No
Serrano <i>et al.</i> (2015)	SQL	HBase	Automático	No
Methakullawat y Limpiyakorn (2014)	PL/SQL	Diagrama de clases	Semiautomático	No
Habringer <i>et al.</i> (2014)	PL/SQL	Diagrama de clases	Automático	No
Egea y Dania (2017)	Diagrama de clases OCL	PL/SQL	Automático	No
Behrend <i>et al.</i> (2009)	PL/SQL	PL/SQL no estándar	No	Sí
Chaverra (2011) Zapata <i>et al.</i> (2011) Zapata <i>et al.</i> (2012)	Esquemas preconceptuales	SQL JSP PHP	Sí	No
Zapata (2012) Noreña (2013) Noreña <i>et al.</i> (2019) Zapata <i>et al.</i> (2014)	Esquemas preconceptuales	Tarjetas de educación Diagrama de procesos Máquinas de estado Grafo de interacción de eventos	Semiautomático	Sí

3. Planteamiento del problema

3.1. Formulación de la hipótesis

Es posible obtener sentencias del lenguaje PL/SQL a partir de la representación de eventos disparadores y de resultado que se realizan en los esquemas preconceptuales.

3.2. Preguntas de investigación

¿Cuáles representaciones de los eventos en el esquema preconceptuales permiten generar sentencias PL/SQL?

¿Se puede automatizar la generación de sentencias?

3.3. Formulación del problema

De acuerdo con los antecedentes, se encuentra en la literatura especializada una tendencia hacia la generación de código en diferentes lenguajes como SQL, JSP y PHP (Chaverra, 2011). También, se encuentran proyectos relacionados con la generación de modelos a partir de código PL/SQL, lo que se constituye como un antecedente directo de esta Tesis de Maestría, pero en sentido contrario al planteamiento general del problema. Así, el problema de esta Tesis de Maestría se centra en el hecho de que en ninguno de los trabajos mencionados se toman en consideración los eventos que se representan en los esquemas preconceptuales, de modo que se está obviando la posibilidad de incorporar el comportamiento de los eventos en el código fuente generado; también, se puede concluir que esa generación es posible dado que ya se realizó previamente el proceso inverso (Methakullawat & Limpiyakorn, 2014). Además, existen nuevas estructuras para la representación de eventos en esquemas preconceptuales, lo cual se liga con el trabajo

actual del grupo de investigación en Lenguajes Computacionales que da sustento a esta Tesis de Maestría.

3.4. Justificación

Breu *et al.* (2011) proponen diez principios para la ingeniería dirigida por el cambio y particularmente enfocada en los modelos vivientes.

- P1. Entornos de modelado centrados en el interesado.
- P2. Acoplamiento cercano entre modelos y sistemas en ejecución.
- P3. Flujo bidireccional de información entre modelos y código.
- P4. Vista común del sistema.
- P5. Persistencia.
- P6. Consistencia y recuperación de la información.
- P7. Dominios y responsabilidades.
- P8. Estados de los elementos del modelo.
- P9. Eventos de cambio.
- P10. Proceso dirigido al cambio.

El primero y el cuarto principio ayudan a justificar la elección de los esquemas preconceptuales como punto de partida, dado su cercanía con el lenguaje natural del interesado y la posibilidad de tener una visión completa del sistema que soporte la comprensión del interesado en relación con el sistema en construcción. El segundo y tercer principio proveen parámetros que permiten definir la trazabilidad completa entre los modelos y el código ejecutable. Con base en el análisis de la propuesta de Methakullawat y Limpiyakorn (2014), se puede evidenciar la necesidad de este acoplamiento para PL/SQL, pero ellos mismos afirman que los sistemas basados en PL/SQL usualmente son sistemas heredados que ya no tienen una documentación adecuada. Por otra parte, Chaverra (2011), Zapata *et al.* (2012) y Zapata *et al.* (2011) permiten deducir que es posible la generación de código desde esquemas preconceptuales, incluyendo código procedimental basado en PHP y código declarativo basado en SQL. Por ello, es posible deducir que PL/SQL, un lenguaje que tiene raíces en ambos formalismos (procedimental y declarativo) también es posible generarlo a partir de esquemas preconceptuales. El caso

de estudio para aplicación de las reglas heurísticas permite evaluar los principios quinto y sexto, en tanto que los principios séptimo a décimo, relacionados todos ellos con el cambio y su necesidad de ser la directriz fundamental en el proceso de desarrollo, se atienden con las reglas heurísticas para la generación semiautomática del código SQL a partir de los eventos del esquema preconceptual. Finalmente, con base en los trabajos de Noreña (2013) y Zapata, Noreña y Vargas (2014) se establecen en los esquemas preconceptuales nuevas estructuras para la representación de eventos de resultado y disparadores. Con un análisis previo de esas estructuras se puede determinar la similitud que tienen los eventos disparadores con la filosofía de los triggers que se definen en PL/SQL.

En síntesis, la solución que se propone en esta Tesis de Maestría tiene su justificación plena en los modelos vivientes, una forma de mantener consistente y rastreable la idea del interesado, expresada desde sus primeros discursos, hasta el código ejecutable. Los esquemas preconceptuales que incluyen los eventos y su especificación constituyen el punto de partida, las reglas heurísticas proporcionan la idea de la bidireccionalidad de la información y los *triggers* definitivos que se obtienen permiten la ejecución en un entorno real para ejemplificar una y otra vez a los interesados la funcionalidad esperada en el proceso de desarrollo.

3.5. Objetivos

3.5.1. General

Formular un conjunto de reglas heurísticas que posibilite la transformación semiautomática de eventos representados en esquemas preconceptuales en sentencias PL/SQL.

3.5.2. Específicos

- Definir las representaciones de eventos en esquemas preconceptuales que son susceptibles de traducir en sentencias PL/SQL
- Establecer las sentencias PL/SQL que tienen correspondencia directa con el tipo de elementos que se establecen en los esquemas preconceptuales.

- Plantear el conjunto de reglas heurísticas que posibilita la transformación semiautomática de eventos representados en esquemas preconceptuales en sentencias PL/SQL.
- Implementar las reglas en un prototipo funcional que permita ejemplificar su uso.
- Validar la transformación propuesta con base en piezas de código PL/SQL reportadas en la literatura científica.

4.Propuesta de Solución

4.1. Definición de reglas heurísticas

Las reglas heurísticas para la generación semiautomática de código PL/SQL a partir de esquemas preconceptuales tienen la estructura típica de una regla, es decir son pares antecedente-consecuente, en los que el antecedente dispara la regla y el consecuente es la postcondición que se logra cuando la regla se ejecute. Los antecedentes son patrones que se pueden detectar en los esquemas preconceptuales con base en sus diferentes elementos (conceptos, relaciones eventuales, eventos, variables, parámetros, valores, operadores, asignaciones, condicionales, etc.) que se equiparan con la sintaxis gráfica del esquema preconceptual. Los consecuentes son plantillas del código a desarrollar, reemplazando los valores consignados en los antecedentes. A las reglas se les asigna una numeración consecutiva, con el fin poder identificar la aplicación del patrón correspondiente en el Capítulo siguiente, en el cual se valida el conjunto de reglas heurísticas con un caso de estudio. Igualmente, se añade un comentario en lenguaje natural a cada regla para aclarar el sentido del patrón.

El conjunto completo de reglas se incluye en las Tablas 4-1 a 4-11.

Tabla 4-1: Reglas heurísticas de transformación #1 y #2

Fuente: Elaboración propia

#	Antecedente	Consecuente
1	<p style="text-align: center;">CONDICIONES INICIALES</p>	<pre>CREATE TABLE "VARIABLE" ("A" NUMBER NOT NULL ENABLE, "B" VARCHAR2(50) NOT NULL ENABLE, "C" NUMBER GENERATED ALWAYS AS ("A"+1) VIRTUAL); INSERT INTO VARIABLE (A, B) VALUES (1, 'VALOR');</pre>
<p>Las variables que se encuentran en las condiciones iniciales del esquema preconceptual se deben agrupar en una tabla denominada "VARIABLE" para evitar el manejo de variables globales en el entorno de desarrollo. Así, se generan las sentencias DDL CREATE TABLE con los tipos de datos correspondientes a NUMBER si la variable es numérica y VARCHAR2(50) si es alfanumérica. Las variables autocalculadas se convierten en campos generados de esta tabla. Igualmente se genera la sentencia DML INSERT INTO para asignar los nombres a las variables.</p>		
	<p style="text-align: center;">CONDICIONES INICIALES</p>	<pre>CREATE TABLE "PARAMETRO" ("A" NUMBER NOT NULL ENABLE, "B" VARCHAR2(50) NOT NULL ENABLE, "C" NUMBER GENERATED ALWAYS AS ("A"+1) VIRTUAL); INSERT INTO PARAMETRO (A, B) VALUES (1, 'VALOR');</pre>
<p>Los parámetros que se encuentran en las condiciones iniciales del esquema preconceptual se deben agrupar en una tabla denominada "PARAMETRO" para evitar el manejo de variables globales en el entorno de desarrollo. Así, se generan las sentencias DDL CREATE TABLE con los tipos de datos correspondientes a NUMBER si el parámetro es numérico y VARCHAR2(50) si es alfanumérico. Los parámetros autocalculados se convierten en campos generados de esta tabla. Igualmente se genera la sentencia DML INSERT INTO para asignar los nombres a las variables.</p>		

Tabla 4-2: Reglas heurísticas de transformación #3 y #4

Fuente: Elaboración propia

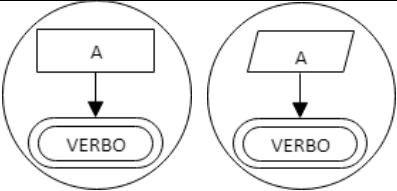
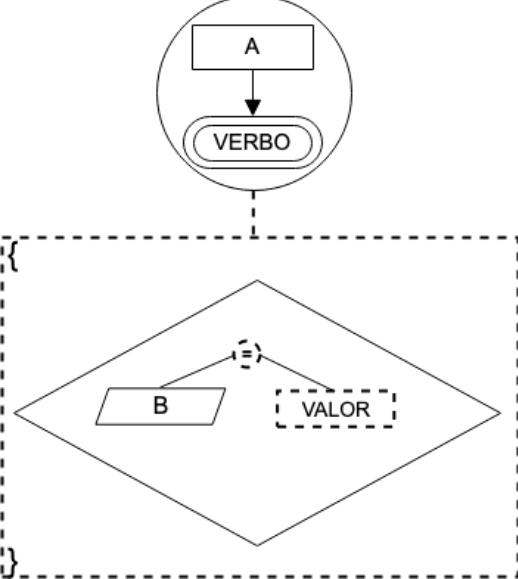
#	Antecedente	Consecuente
3		<pre>CREATE [OR REPLACE] TRIGGER -"A_VERBO"</pre>
<p>El nombre del <i>trigger</i> se toma directamente de la imagen del evento, como una combinación del nombre del nodo (ya sea concepto o variable) y el verbo correspondiente a la relación eventual.</p>		
4		<pre>CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "B" ON ["VARIABLE"] DECLARE "v_" + B [VARIABLE].B%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; IF "v_" + B [CONDICION] VALOR THEN END IF; END;</pre>
<p>El primer condicional de la restricción asociada con el evento permite definir la condición de disparo del <i>trigger</i>. Se crea la especificación del disparo en la sección AFTER, se crea una variable que se encabeza con el texto "v_" en la sección de declaración y se genera tanto el SELECT como la condición IF en el cuerpo del <i>trigger</i>.</p>		

Tabla 4-3: Regla heurística de transformación #5

Fuente: Elaboración propia

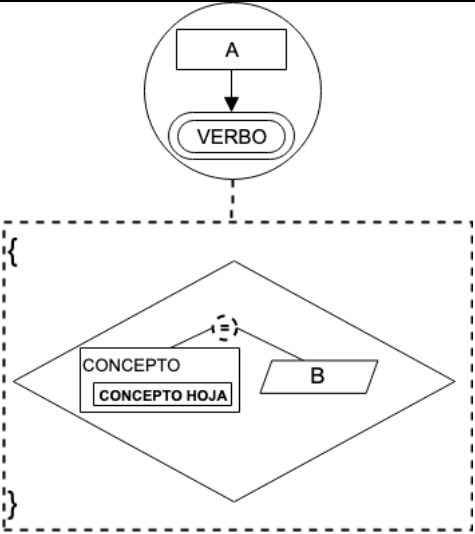
#	Antecedente	Consecuente
5		<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "CONCEPTO_HOJA" ON ["CONCEPTO"] DECLARE "v_" + CONCEPTO_HOJA [CONCEPTO].CONCEPTO_HOJA%TYPE; "v_" + B [VARIABLE/"PARAMETRO"].B%TYPE; BEGIN SELECT CONCEPTO_HOJA INTO "v_" + CONCEPTO_HOJA FROM [CONCEPTO]; SELECT B INTO "v_" + B FROM [VARIABLE/PARAMETRO]; IF "v_" + CONCEPTO_HOJA [CONDICION] "v_" + B THEN END IF; END; </pre>
<p>Esta regla es análoga a la regla #4, pero con un concepto compuesto asociado con el condicional. En la declaración hay que crear todos los elementos que hagan parte del condicional y en el cuerpo se aplican las sentencias SELECT para todos los elementos.</p>		

Tabla 4-4: Regla heurística de transformación #6

Fuente: Elaboración propia

#	Antecedente	Consecuente
6	<p>The diagram illustrates the transformation of a trigger condition. It is divided into two parts. The top part shows a trigger 'A' with the verb 'VERBO' leading to a condition 'B < VALOR_1'. Below this, a dashed box contains a condition 'C = VALOR_2'. The bottom part shows the same trigger 'A' with the verb 'VERBO' leading to a condition 'B < VALOR_2'.</p>	<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "B" ON ["VARIABLE"] DECLARE "v_" + B [VARIABLE].B%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; IF v_B [CONDICION] VALOR_1 THEN UPDATE [VARIABLE] SET C='VALOR_2'; END IF; END; </pre>
<p>Las asignaciones que se hagan después del condicional de un evento deben aparecer explícitamente como actualizaciones a la tabla VARIABLE, ya sea que se encuentren dentro de una restricción o que simplemente finalicen la restricción donde está el condicional.</p>		

Tabla 4-5: Regla heurística de transformación #7

Fuente: Elaboración propia

#	Antecedente	Consecuente
7	<p>The diagram illustrates the antecedent of rule #7. At the top, a circle contains a box labeled 'A' with an arrow pointing down to a rounded rectangle labeled 'VERBO'. Below this, a dashed line leads to a diamond-shaped restriction containing a condition 'B > VALOR_1'. An arrow points from this diamond to a dashed box containing two conditions: 'C = VALOR_2' and 'B = VALOR_3'.</p>	<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "B" ON ["VARIABLE"] DECLARE "v_" + B [VARIABLE].B%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; IF "v_" + B [CONDICION] VALOR_1 THEN UPDATE [VARIABLE] SET C='VALOR_2'; END IF; END; CREATE [OR REPLACE] TRIGGER "A_VERBO" + "_1" AFTER [UPDATE/INSERT] OF "C" ON ["VARIABLE"] BEGIN UPDATE [VARIABLE] SET B='VALOR_3'; END; </pre>
<p>En caso de que en la restricción asociada con el evento se intente hacer la actualización del mismo campo (variable o concepto-hoja) que se usa en la condición, se genera el fenómeno de tabla mutante. En este caso se generan dos <i>triggers</i> enlazados mediante la condición final de la restricción inmediatamente ligada con la condición (en este caso C=VALOR), que se convierte en la condición de disparo del segundo <i>trigger</i>. El resto de la regla se aplica de la misma manera que la regla #6</p>		

Tabla 4-6: Regla heurística de transformación #8

Fuente: Elaboración propia

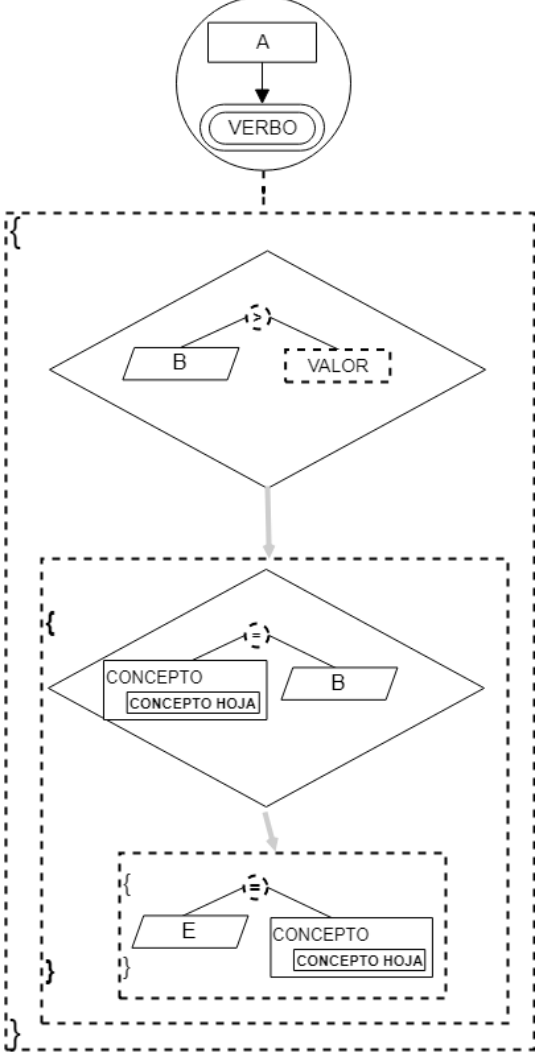
#	Antecedente	Consecuente
8	 <p>The diagram illustrates the antecedent of rule #8. It starts with a node 'A' in a rounded rectangle, which points to a rounded rectangle labeled 'VERBO'. Below this, a dashed box contains a diamond-shaped decision node with two outgoing paths: one to a parallelogram labeled 'B' and another to a dashed box labeled 'VALOR'. This diamond is connected to a second diamond-shaped decision node, also within a dashed box. This second diamond has two paths: one to a rounded rectangle containing 'CONCEPTO' and 'CONCEPTO HOJA' stacked vertically, and another to a parallelogram labeled 'B'. This second diamond is connected to a third diamond-shaped decision node, also within a dashed box. This third diamond has two paths: one to a parallelogram labeled 'E' and another to a rounded rectangle containing 'CONCEPTO' and 'CONCEPTO HOJA' stacked vertically.</p>	<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "B" ON ["VARIABLE"] DECLARE "v_" + B [VARIABLE].B%TYPE; "v_" + CONCEPTO_HOJA [CONCEPTO].CONCEPTO_HOJA%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; SELECT CONCEPTO_HOJA INTO "v_" + CONCEPTO_HOJA FROM [CONCEPTO]; IF "v_" + B [CONDICION] VALOR THEN IF "v_" + CONCEPTO_HOJA [CONDICION] "v_" + B THEN UPDATE [VARIABLE] SET E= "v_" + CONCEPTO_HOJA; END IF; END IF; END; </pre>
<p>Esta regla es una ampliación de la regla #5 para el caso en que se use un concepto compuesto para llevar su valor a una variable de manera temporal. La actualización se realiza en una sentencia IF anidada que incluye los dos condicionales.</p>		

Tabla 4-7: Regla heurística de transformación #9

Fuente: Elaboración propia

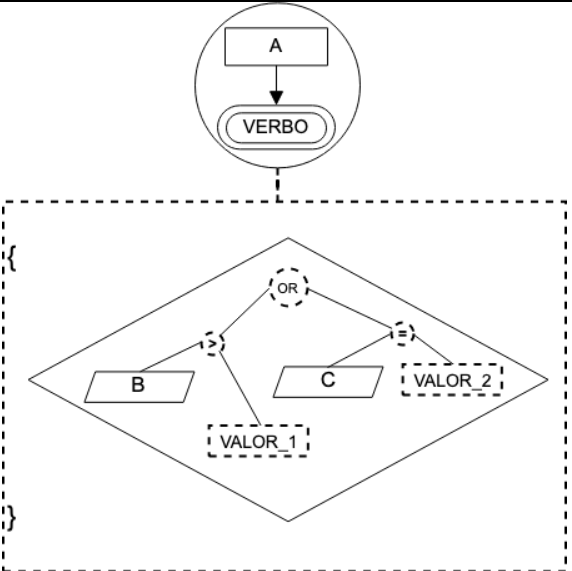
#	Antecedente	Consecuente
9		<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" + "_B" AFTER [UPDATE/INSERT] OF "B" ON ["VARIABLE"] DECLARE "v_" + B [VARIABLE].B%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; IF "v_" + B [CONDICION] VALOR_1 THEN END IF; END; CREATE [OR REPLACE] TRIGGER "A_VERBO" + "_C" AFTER [UPDATE/INSERT] OF "C" ON ["VARIABLE"] DECLARE "v_" + C [VARIABLE].C%TYPE; BEGIN SELECT C INTO "v_" + V FROM [VARIABLE]; IF "v_" + C [CONDICION] VALOR_2 THEN END IF; END; </pre>
<p>Un operador de comparación OR en el condicional de un evento se resuelve separando las condiciones para dos <i>triggers</i> independientes.</p>		

Tabla 4-8: Regla heurística de transformación #10

Fuente: Elaboración propia

#	Antecedente	Consecuente
10		<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "B" ON ["VARIABLE"] DECLARE "v_" + B [VARIABLE].B%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; IF "v_" + B [CONDICION] VALOR_1 THEN [UPDATE/INSERT] "C" SET D=VALOR_2; END IF; END; </pre>
<p>Una relación dinámica atómica INSERTA al interior de la restricción de un evento genera una inserción o actualización en el concepto-hoja de una tabla.</p>		

Tabla 4-9: Regla heurística de transformación #11

Fuente: Elaboración propia

#	Antecedente	Consecuente
11		<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "B" ON ["VARIABLE"] DECLARE "v_" + B [VARIABLE].B%TYPE; "v_" + C [VARIABLE].C%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; IF v_B [CONDICION] VALOR_1 THEN UPDATE [VARIABLE] SET C='VALOR_2'; END IF; END; CREATE [OR REPLACE] TRIGGER "D_VERBO" AFTER [UPDATE/INSERT] OF "C" ON ["VARIABLE"] DECLARE "v_" + C [VARIABLE].C%TYPE; BEGIN SELECT C INTO "v_" + C FROM [VARIABLE]; IF v_C [CONDICION] VALOR_3 THEN END IF; END; </pre>
<p>Con esta regla, más que una transformación se pretende una validación de la implicación entre eventos. Para que la implicación se presente, el elemento que se actualiza (variable o concepto hoja) en el evento de partida se debe invocar en el condicional de disparo del evento de llegada.</p>		

Tabla 4-10: Regla heurística de transformación #12

Fuente: Elaboración propia

#	Antecedente	Consecuente
12		<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "B" ON ["VARIABLE"] DECLARE "v_" + B [VARIABLE].B%TYPE; "v_" + C [VARIABLE].C%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; IF v_B [CONDICION] VALOR_1 THEN UPDATE [VARIABLE] SET C=VALOR_2; END IF; END; CREATE [OR REPLACE] TRIGGER "D_VERBO" AFTER [UPDATE/INSERT] OF "E" ON ["VARIABLE"] DECLARE "v_" + E [VARIABLE].E%TYPE; "v_" + F [VARIABLE].F%TYPE; BEGIN SELECT E INTO "v_" + E FROM [VARIABLE]; IF v_E [CONDICION] VALOR_3 THEN UPDATE [VARIABLE] SET F=VALOR_4; END IF; END; CREATE [OR REPLACE] TRIGGER "G_VERBO" AFTER [UPDATE/INSERT] OF "C" OR AFTER [UPDATE/INSERT] OF "F" ON ["VARIABLE"] DECLARE "v_" + C [VARIABLE].C%TYPE; "v_" + F [VARIABLE].F%TYPE; BEGIN SELECT C INTO "v_" + C FROM [VARIABLE]; SELECT F INTO "v_" + F FROM [VARIABLE]; IF v_C [CONDICION] VALOR_5 AND v_F [CONDICION] VALOR_6 THEN END IF; END; </pre>
<p>En el caso de una relación OR asociada con implicaciones entre eventos (léase A_VERBO o D_VERBO implican G_VERBO), la condición de disparo del tercer <i>trigger</i> (G_VERBO) debe ser la disyunción de la inserción/actualización de ambos elementos (C o F).</p>		

Tabla 4-11: Regla heurística de transformación #13

Fuente: Elaboración propia

#	Antecedente	Consecuente
13	<p>The diagram illustrates the transformation of two triggers into a single one. At the top, two triggers are shown: 'A' (VERBO 1) and 'D' (VERBO 2). Trigger A has a condition on variable B (VALOR_1) and another on variable C (VALOR_2). Trigger D has a condition on variable E (VALOR_3) and another on variable F (VALOR_4). A thick arrow points down to a single trigger 'G' (VERBO 3). Trigger G has a condition on variable C (VALOR_5) and another on variable F (VALOR_6), with an AND operator between them. Dashed boxes group the conditions of the original triggers and the resulting trigger.</p>	<pre> CREATE [OR REPLACE] TRIGGER "A_VERBO" AFTER [UPDATE/INSERT] OF "B" ON [VARIABLE] DECLARE "v_" + B [VARIABLE].B%TYPE; "v_" + C [VARIABLE].C%TYPE; BEGIN SELECT B INTO "v_" + B FROM [VARIABLE]; IF v_B [CONDICION] VALOR_1 THEN UPDATE [VARIABLE] SET C=VALOR_2; END IF; END; CREATE [OR REPLACE] TRIGGER "D_VERBO" AFTER [UPDATE/INSERT] OF "E" ON [VARIABLE] DECLARE "v_" + E [VARIABLE].E%TYPE; "v_" + F [VARIABLE].F%TYPE; BEGIN SELECT E INTO "v_" + E FROM [VARIABLE]; IF v_E [CONDICION] VALOR_3 THEN UPDATE [VARIABLE] SET F=VALOR_4; END IF; END; CREATE [OR REPLACE] TRIGGER "G_VERBO" AFTER [UPDATE/INSERT] OF "C" AND [UPDATE/INSERT] OF "F" ON [VARIABLE] DECLARE "v_" + C [VARIABLE].C%TYPE; "v_" + F [VARIABLE].F%TYPE; BEGIN SELECT C INTO "v_" + C FROM [VARIABLE]; SELECT F INTO "v_" + F FROM [VARIABLE]; IF v_C [CONDICION] VALOR_5 AND v_F [CONDICION] VALOR_6 THEN END IF; END; </pre>
<p>En esta regla, las condiciones son muy similares a la regla# 12, pero en este caso se exige la inserción/actualización de ambos elementos (C y F).</p>		

4.2. Consideraciones especiales en relación con las reglas heurísticas

Feuerstein (2008) habla sobre los peligros de las variables globales en PL/SQL cuando afirma que se deberían evitar porque crean dependencias ocultas o efectos colaterales. Especialmente cuando se trabaja con *triggers*, las variables globales afectan el desarrollo del proceso y pueden causar pérdidas y errores de compilación en la construcción del *trigger*. Una de las maneras en que se recomienda solucionar el problema de las variables globales (representadas en esta Tesis de Maestría con las variables y los parámetros que hacen parte de las condiciones iniciales) se relaciona con su conversión a un esquema de persistencia, que es la solución que se propone en las reglas heurísticas de transformación. Por esta razón, las condiciones iniciales y los consecuentes valores que se asignan a variables y parámetros se convierten en dos tablas separadas que contienen la información correspondiente. En la tabla variables se introducen en forma de columnas todos los valores que tienen susceptibilidad de cambiar durante la ejecución del sistema, en tanto que en parámetros se introducen como columnas todos los valores que permanecen constantes durante la ejecución. De esta forma, también se lidia con varios de los problemas temporales que mencionan Behrend *et al.* (2009), pues la ejecución de los *triggers* se controla mediante actualizaciones de tablas y en este caso la tabla variable es una de las que usualmente más cambia en tiempo de ejecución.

Feuerstein (2012) establece que un error común cuando se realizan aplicaciones con PL/SQL se relaciona con las denominadas tablas mutantes. El error se identifica con el código ORA-4091 y ocurre cuando un *trigger* a nivel de fila trata de examinar o cambiar una tabla que ya se encuentra cambiando, por ejemplo con un INSERT, un UPDATE o un DELETE. Si bien ese error se puede evitar desde el diseño cuando se trata de valores asociados con tablas del modelo, en el caso de los esquemas preconceptuales no se espera que las variables y los parámetros se conviertan usualmente en tablas, pero por la justificación del fenómeno de las variables globales, es necesario que se manejen así. Por ello, el fenómeno de las tablas mutantes se vuelve más difícil de evitar desde el diseño y,

por ello, se incluye la regla heurística #7 para que se haga una separación del evento en dos *triggers*, sin que se afecte la lógica del diseñador.

Una última consideración se relaciona con la regla heurística #9, en la cual también se preserva la lógica del diseñador en la definición del evento, pero la materialización se realiza con dos *triggers* independientes, pero esta vez por la presencia de un operador lógico OR al interior del condicional. La presencia de este tipo de reglas deja en libertad al diseñador para que implemente su lógica como mejor se adecúe al problema, sin tener limitaciones que lo aten finalmente al lenguaje de programación.

5. Validación de la propuesta de solución

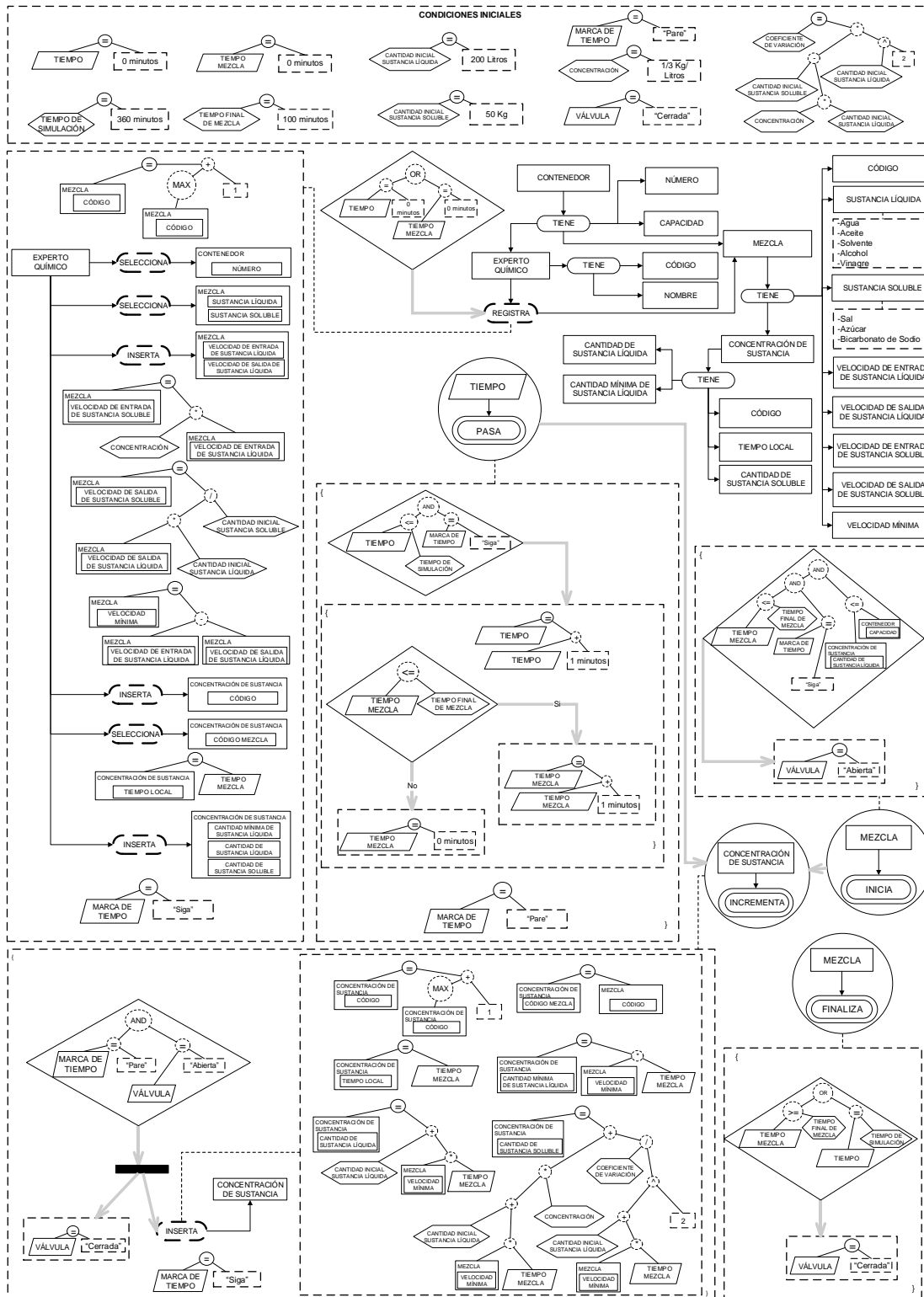
Para realizar la validación de las reglas heurísticas de transformación que se presentan en el Capítulo anterior, se parte de un esquema preconceptual que incluye eventos y que proponen Noreña *et al.* (2019). El caso se describe en la siguiente subsección y luego se muestra su aplicación paso a paso. El entorno seleccionado para la aplicación de las reglas es *Oracle® Application Express*, que es una herramienta de desarrollo rápido de aplicaciones con una base de datos Oracle® que, aunque limitada en almacenamiento, posee la funcionalidad plena de una base de datos completa. Además, se puede usar directamente en un entorno web (<https://apex.oracle.com/en/>) con una cuenta universitaria que se solicita al administrador del sitio. Finalmente, el soporte pleno del lenguaje PL/SQL hace de este entorno la herramienta indicada para realizar la validación de esta Tesis de Maestría.

5.1. Definición del caso de estudio

El caso de estudio que se plantea para validar las reglas heurísticas de transformación de esquemas preconceptuales a código PL/SQL se basa en el esquema de la Figura 5-1 que proponen Noreña *et al.* (2019), en relación con la determinación de la concentración de una mezcla que tiene una sustancia líquida y una sustancia soluble. El esquema incluye un conjunto de condiciones iniciales con variables y parámetros que afectan el fenómeno en estudio, una operación que realiza un experto químico y que da inicio al análisis de la concentración de sustancia en el contenedor y un conjunto de eventos que le dan automatismo al proceso: el paso del tiempo, el inicio de la mezcla, el incremento de la concentración y la finalización de la mezcla.

Figura 5-1: Esquema preconceptual del caso de estudio

Fuente: Noreña *et al.* (2019)



5.2. Aplicación paso a paso de las reglas heurísticas de transformación y funcionamiento del prototipo

Como punto de partida es necesario aplicar las reglas de Chaverra (2011) para la generación de la estructura de datos a partir del esquema preconceptual. La estructura generada representa el siguiente código SQL:

```
CREATE TABLE "CONTENEDOR"
(
  "NUMERO" NUMBER NOT NULL ENABLE,
  "CAPACIDAD" NUMBER NOT NULL ENABLE,
  "CODIGO_EXPERTO" NUMBER,
  CONSTRAINT "CONTENEDOR_PK" PRIMARY KEY ("NUMERO") USING INDEX ENABLE
)
```

```
CREATE TABLE "EXPERTO_QUIMICO"
(
  "CODIGO" NUMBER NOT NULL ENABLE,
  "NOMBRE" VARCHAR2(50) NOT NULL ENABLE
)
```

```
CREATE TABLE "MEZCLA"
(
  "CODIGO" NUMBER NOT NULL ENABLE,
  "SUSTANCIA_LIQUIDA" VARCHAR2(50) NOT NULL ENABLE,
  "SUSTANCIA_SOLUBLE" VARCHAR2(50) NOT NULL ENABLE,
  "VELOCIDAD_ENTRADA_SUSTANCIA_LI" NUMBER NOT NULL ENABLE,
  "VELOCIDAD_SALIDA_SUSTANCIA_LIQ" NUMBER NOT NULL ENABLE,
  "VELOCIDAD_ENTRADA_SUSTANCIA_SO" NUMBER NOT NULL ENABLE,
  "VELOCIDAD_SALIDA_SUSTANCIA_SOL" NUMBER NOT NULL ENABLE,
  "VELOCIDAD_MINIMA" NUMBER NOT NULL ENABLE,
  "CONTENEDOR" NUMBER NOT NULL ENABLE,
  CONSTRAINT "SUSTANCIA_LIQUIDA_PERMITIDA" CHECK ( "SUSTANCIA_LIQUIDA" IN
('AGUA', 'ACEITE', 'SOLVENTE', 'ALCOHOL', 'VINAGRE')) ENABLE,
  CONSTRAINT "SUSTANCIA_SOLUBLE_PERMITIDA" CHECK ( "SUSTANCIA_SOLUBLE" IN
('SAL', 'AZUCAR', 'BICARBONATO_DE_SODIO')) ENABLE
)
```

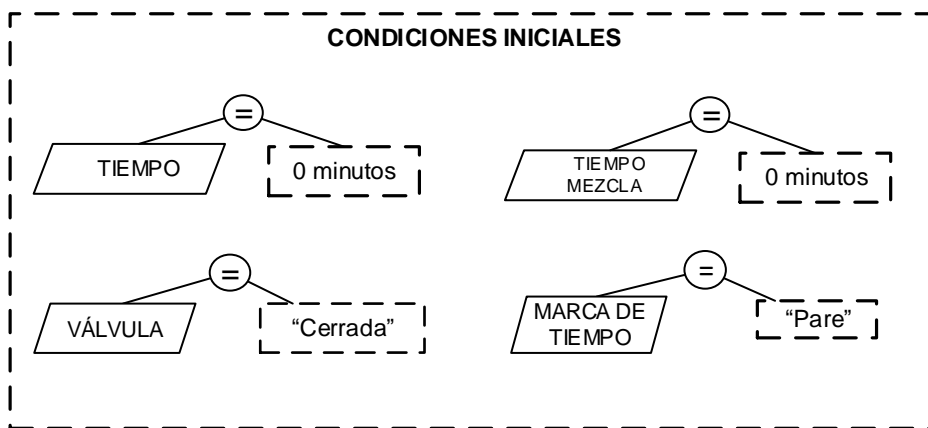
```
CREATE TABLE "CONCENTRACION_DE_SUSTANCIA"
(
```

```
"CODIGO" NUMBER NOT NULL ENABLE,  
"CANTIDAD_DE_SUSTANCIA_LIQUIDA" NUMBER,  
"CANTIDAD_MINIMA_DE_SUSTANCIA_L" NUMBER,  
"CANTIDAD_DE_SUSTANCIA_SOLUBLE" NUMBER NOT NULL ENABLE,  
"CODIGO_MEZCLA" NUMBER,  
"TIEMPO_LOCAL" NUMBER NOT NULL ENABLE  
)
```

Se debe anotar que las restricciones de la tabla concentración de sustancia (es decir, SUSTANCIA_LIQUIDA_PERMITIDA y SUSTANCIA_SOLUBLE_PERMITIDA) se generan manualmente, pues las reglas de Chaverra (2011) no abarcan restricciones. En la Figura 5-2 se presentan las condiciones iniciales relacionadas con las variables del caso de estudio.

Figura 5-2: Condiciones iniciales de las variables del caso de estudio

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



A partir de la aplicación de la regla #1, con las variables dentro de las condiciones iniciales se genera una tabla llamada VARIABLE con la respectiva estructura. Además, se genera un INSERT con los respectivos valores asignados, como se muestra en el código siguiente:

```
CREATE TABLE "VARIABLE"  
(  
  "TIEMPO" NUMBER NOT NULL ENABLE,  
  "TIEMPO_MEZCLA" NUMBER NOT NULL ENABLE,  
  "MARCA_DE_TIEMPO" VARCHAR2(50) NOT NULL ENABLE,
```

```
"VALVULA"          VARCHAR2(50) NOT NULL ENABLE
);
```

```
INSERT INTO VARIABLE (TIEMPO, TIEMPO_MEZCLA, MARCA_DE_TIEMPO, VALVULA)
VALUES (0, 0, 'PARE', 'CERRADA');
```

Nótese que los tipos de datos de las variables son NUMBER o VARCHAR2(50) y se obtienen según sea el tipo de dato del cual se asigna la variable. Los nombres de las variables solo pueden tener una extensión máxima de 30 *bytes* por restricción del lenguaje; por lo tanto, si el nombre contiene más caracteres de lo permitido, se debe limitar hasta donde cumpla la condición. Al final, al interior del *Oracle® Application Express*, la tabla VARIABLE posee la estructura que se muestra en la Figura 5-3. Los datos que se insertan en la tabla, se muestran en la Figura 5-4.

Figura 5-3: Estructura de la tabla VARIABLE

Fuente: Elaboración propia

Table	Data	Indexes	Model	Constraints	Grants	Statistics	UI Defaults	Triggers	Dependencies	SQL	REST
<div style="display: flex; justify-content: space-between;"> Add Column Modify Column Rename Column Drop Column Rename Copy Drop Truncate Create Lookup Table </div>											
Column Name	Data Type										
TIEMPO	NUMBER										No
TIEMPO_MEZCLA	NUMBER										No
MARCA_DE_TIEMPO	VARCHAR2(50)										No
VALVULA	VARCHAR2(50)										No

Figura 5-4: Valores asignados a la tabla VARIABLE

Fuente: Elaboración propia

EDIT	TIEMPO	TIEMPO_MEZCLA	MARCA_DE_TIEMPO	VALVULA
<input type="checkbox"/>	0	0	PARE	CERRADA

En la Figura 5-5 se presentan las condiciones iniciales relacionadas con los parámetros del caso de estudio.


```
"COEFICIENTE_DE_VARIACION"          NUMBER generated always AS
(("CANTIDAD_INICIAL_SUSTANCIA_LIQ"*"CANTIDAD_INICIAL_SUSTANCIA_LIQ")*
("CANTIDAD_INICIAL_SUSTANCIA_SOL" -
("CONCENTRACION"*"CANTIDAD_INICIAL_SUSTANCIA_SOL"))) VIRTUAL
);
```

```
INSERT INTO PARAMETRO (TIEMPO_DE_SIMULACION, TIEMPO_FINAL_DE_MEZCLA,
CANTIDAD_INICIAL_SUSTANCIA_LIQ, CANTIDAD_INICIAL_SUSTANCIA_SOL,
CONCENTRACION)
VALUES (14, 12, 200, 50, 0.3333333333 );
```

Al igual que en el caso de las variables, los nombres de los parámetros también poseen la restricción de 30 bytes de extensión máxima. Para los parámetros que requieran la construcción de alguna estructura para generar automáticamente el valor, es necesario el uso del comando VIRTUAL, con el cual se especifica que dicha columna posee un contenido basado en una fórmula o expresión usando datos de otras columnas. La estructura final de la tabla PARAMETRO se muestra en la Figura 5-6. El resultado de la sentencia INSERT se muestra en la Figura 5-7. Además, en la Figura 5-8 se muestra la definición gráfica del primer evento denominado TIEMPO PASA.

Figura 5-6: Estructura de la tabla PARÁMETRO

Fuente: Elaboración propia

Column Name	Data Type	Nullable	Default
TIEMPO_DE_SIMULACION	NUMBER	No	-
TIEMPO_FINAL_DE_MEZCLA	NUMBER	No	-
CANTIDAD_INICIAL_SUSTANCIA_LIQ	NUMBER	No	-
CANTIDAD_INICIAL_SUSTANCIA_SOL	NUMBER	No	-
CONCENTRACION	NUMBER	No	-
COEFICIENTE_DE_VARIACION	NUMBER	Yes	"CANTIDAD_INICIAL_SUSTANCIA_LIQ"*"CANTIDAD_INICIAL_SUSTANCIA_LIQ"/("CANTIDAD_INICIAL_SUSTANCIA_SOL"-("CONCENTRACION"*"CANTIDAD_INICIAL_SUSTANCIA_SOL"))

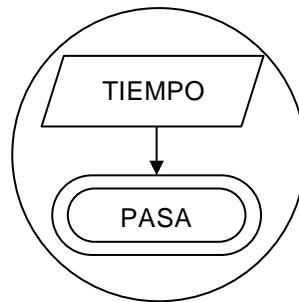
Figura 5-7: Valores asignados a la tabla PARÁMETRO

Fuente: Elaboración propia

EDIT	TIEMPO_DE_SIMULACION	TIEMPO_FINAL_DE_MEZCLA	CANTIDAD_INICIAL_SUSTANCIA_LIQ	CANTIDAD_INICIAL_SUSTANCIA_SOL	CONCENTRACION	COEFICIENTE_DE_VARIACION
	14	12	200	50	.3333333333	133333.33334

Figura 5-8: Notación gráfica del evento TIEMPO PASA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)

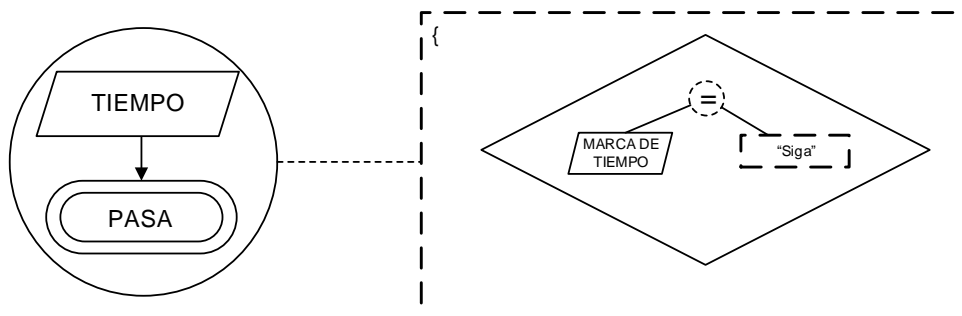


Al aplicar la regla #3 se genera el código inicial del *trigger*. El nombre de los *triggers* también posee la restricción de los 30 bytes. En este caso se trata de un *trigger* sobre la variable tiempo, que se generó como una tabla adicional en la base de datos. En la Figura 5-9 se muestra el condicional inicial del evento. Al aplicar la regla #4 se genera la condición que hace disparar el *trigger*.

```
CREATE OR REPLACE TRIGGER "TIEMPO_PASA"
```

Figura 5-9: Condicional inicial del evento TIEMPO PASA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



```
CREATE OR REPLACE TRIGGER "TIEMPO_PASA"  
AFTER  
    UPDATE OF "MARCA_DE_TIEMPO" ON "VARIABLE"  
DECLARE
```

```

v_marca_de_tiempo VARIABLE.MARCA_DE_TIEMPO%TYPE;
BEGIN
  SELECT MARCA_DE_TIEMPO INTO v_marca_de_tiempo FROM VARIABLE;
  IF v_marca_de_tiempo='SIGA' THEN
  END IF;
END;

```

TIEMPO PASA se dispara en el momento en que la columna MARCA DE TIEMPO de la tabla VARIABLE se actualice. Posteriormente, se genera la declaración de variables. Cada variable o parámetro que se encuentre dentro de la restricción corresponde a una variable dentro del *trigger*. El nombre de estas variables también posee la restricción de los 30 *bytes*. Las variables temporales se nombran usando el nombre de la variable y el prefijo “v_”, con el cual se debe especificar un tipo con base en la columna a la cual representa dicha variable.

```

v_marca_de_tiempo VARIABLE.MARCA_DE_TIEMPO%TYPE;

```

La variable v_marca_de_tiempo representa la columna MARCA_DE_TIEMPO de la tabla VARIABLE que posee tipo VARCHAR2(50). La condición de disparo del *trigger* también debe poseer un condicional que verifique si dicha condición si se cumple.

```

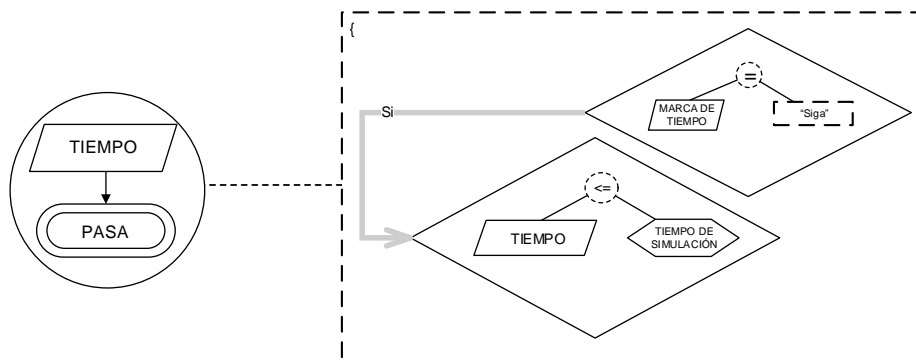
IF v_marca_de_tiempo='SIGA' THEN
END IF;

```

En el *trigger* se debe preguntar si la variable MARCA_DE_TIEMPO tiene un valor que corresponda a SIGA para realizar alguna acción. En la Figura 5-10 se muestra un condicional relacionado por medio de implicación con el condicional inicial.

Figura 5-10: Condicional anidado con el condicional inicial del evento TIEMPO PASA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



Paso seguido, se genera el condicional anidado. Para que la sentencia se pueda evaluar se debe aplicar nuevamente la regla #4, en la que se especifica que toda variable o parámetro dentro de la restricción corresponde a una variable dentro del *trigger*, por lo cual es necesario tener variables temporales que soporten TIEMPO y TIEMPO_DE_SIMULACION.

```
CREATE OR REPLACE TRIGGER "TIEMPO_PASA"  
AFTER  
  UPDATE OF "MARCA_DE_TIEMPO" ON "VARIABLE"  
DECLARE  
  v_MARCA_DE_TIEMPO VARIABLE.MARCA_DE_TIEMPO%TYPE;  
  
  v_TIEMPO VARIABLE.TIEMPO%TYPE;  
  v_TIEMPO_DE_SIMULACION PARAMETRO.TIEMPO_DE_SIMULACION%TYPE;  
  
BEGIN  
  SELECT MARCA_DE_TIEMPO INTO v_MARCA_DE_TIEMPO FROM VARIABLE;  
  
  SELECT TIEMPO INTO v_TIEMPO FROM VARIABLE;  
  SELECT TIEMPO_DE_SIMULACION INTO v_TIEMPO_DE_SIMULACION FROM PARAMETRO;  
  
  IF v_MARCA_DE_TIEMPO='SIGA' THEN  
    IF v_TIEMPO<=v_TIEMPO_DE_SIMULACION THEN  
      END IF;  
    END IF;  
END;
```

Una vez definida la estructura se puede generar la siguiente restricción que corresponde a una asignación y a un condicional anidado, que se muestran en la Figura 5-11. Para ello, se aplica la regla #6, en la que se especifica que cualquier asignación de variables dentro de las restricciones corresponde a un UPDATE de dicha columna en la tabla VARIABLE. Siempre que se defina una variable nueva dentro del *trigger*, se debe acompañar con su SELECT respectivo. Las asignaciones realizadas se realizan con el comando UPDATE para editar los valores dentro de la tabla "VARIABLE".

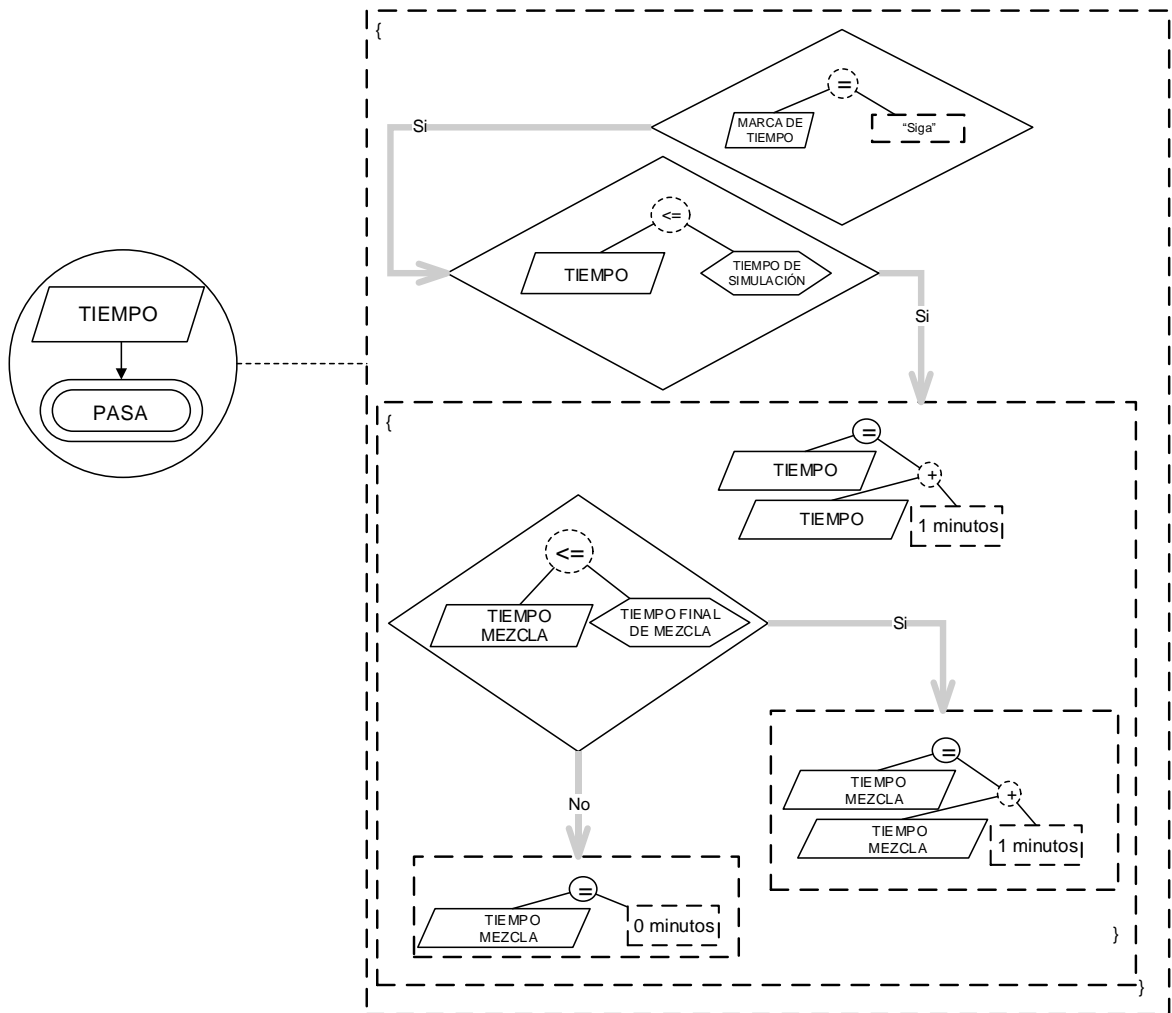
```
UPDATE VARIABLE SET TIEMPO = TIEMPO + 1;
```

Se aplica la regla #1 para la asignación de la columna TIEMPO, la cual, aunque se autogenera, no representa una complejidad de grado alto y no contempla datos de las

demás columnas, por lo cual no es necesario el uso del comando VIRTUAL. En la Figura 5-12 se completa la sintaxis del evento TIEMPO PASA.

Figura 5-11: Restricción con asignación de variables del evento TIEMPO PASA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



```

CREATE OR REPLACE TRIGGER "TIEMPO_PASA"
AFTER
    UPDATE OF "MARCA_DE_TIEMPO" ON "VARIABLE"
DECLARE
    v_MARCA_DE_TIEMPO VARIABLE.MARCA_DE_TIEMPO%TYPE;

    v_TIEMPO VARIABLE.TIEMPO%TYPE;
    v_TIEMPO_DE_SIMULACION PARAMETRO.TIEMPO_DE_SIMULACION%TYPE;
    
```

```
v_TIEMPO_MEZCLA VARIABLE.TIEMPO_MEZCLA%TYPE;
v_TIEMPO_FINAL_DE_MEZCLA PARAMETRO.TIEMPO_FINAL_DE_MEZCLA%TYPE;
BEGIN
  SELECT MARCA_DE_TIEMPO INTO v_MARCA_DE_TIEMPO FROM VARIABLE;

  SELECT TIEMPO INTO v_TIEMPO FROM VARIABLE;
  SELECT TIEMPO_DE_SIMULACION INTO v_TIEMPO_DE_SIMULACION FROM PARAMETRO;

  SELECT TIEMPO_MEZCLA INTO v_TIEMPO_MEZCLA FROM VARIABLE;
  SELECT TIEMPO_FINAL_DE_MEZCLA INTO v_TIEMPO_FINAL_DE_MEZCLA FROM
PARAMETRO;

  IF v_MARCA_DE_TIEMPO='SIGA' THEN
    UPDATE VARIABLE SET TIEMPO = TIEMPO + 1;
    IF v_TIEMPO<=v_TIEMPO_DE_SIMULACION THEN
      IF v_TIEMPO_MEZCLA <= v_TIEMPO_FINAL_DE_MEZCLA THEN

        UPDATE VARIABLE SET TIEMPO_MEZCLA = v_TIEMPO_MEZCLA + 1;
      ELSE
        UPDATE VARIABLE SET TIEMPO_MEZCLA = 0;
      END IF;
    END IF;
  END IF;
END;
```


condición que lo dispara, no es posible actualizar su contenido como lo plantea el esquema, pues, de hacerlo, se generaría un error de tabla mutante, afectando la integridad referencial del evento. En la regla #7 se propone la separación del *trigger* en dos partes: aquel que realiza las acciones reflejadas en el preconceptual y aquella que realice la actualización de sobre la variable que activa el *trigger*, como se muestra a continuación:

```
CREATE TRIGGER "TIEMPO_PASA_1"  
AFTER  
    UPDATE OF "TIEMPO_MEZCLA"  
    ON "VARIABLE"  
BEGIN  
    UPDATE VARIABLE SET MARCA_DE_TIEMPO='PARE';  
END;
```

El nombre del nuevo *trigger* se complementa con el sufijo “_1”. En caso de que el nombre del evento supere la restricción de los 30 *bytes* y no sea posible concatenar el sufijo, se deben eliminar dos caracteres hacia la izquierda para concatenar el sufijo. Como se dicta en la regla, la última actualización que se realice sobre el *trigger* permitirá definir el elemento de disparo para este nuevo *trigger*. En este caso TIEMPO_MEZCLA es el último elemento que se actualiza en el *trigger*, por lo cual la actualización de esta columna será el evento que dispare este nuevo *trigger*. El evento a realizar dentro del *trigger* UPDATE de MARCA_DE_TIEMPO es la acción a realizar una vez el *trigger* se active. Al final de la generación del *trigger* a partir del evento, ya se completa la estructura. La declaración de ambos *triggers* no presenta errores de compilación y se permite su creación. En la Figura 5-13 se muestran los dos *triggers* generados sobre la tabla VARIABLE en el entorno Oracle® Application Express. Además, en la Figura 5-14 se muestra la representación gráfica del siguiente evento a generar, denominado MEZCLA INICIA, para el cual se aplica la regla #3 y se obtiene el código respectivo. En la Figura 5-15 se comienza a detallar el evento.

Figura 5-13: Triggers finalmente incorporados en la tabla VARIABLE

Fuente: Elaboración propia

```

UPDATE OF "MARCA_DE_TIEMPO"
ON "VARIABLE"
DECLARE
  v_MARCA_DE_TIEMPO VARIABLE.MARCA_DE_TIEMPO%TYPE;
  v_TIEMPO VARIABLE.TIEMPO%TYPE;
  v_TIEMPO_DE_SIMULACION PARAMETRO.TIEMPO_DE_SIMULACION%TYPE;
  v_TIEMPO_MEZCLA VARIABLE.TIEMPO_MEZCLA%TYPE;
  v_TIEMPO_FINAL_DE_MEZCLA PARAMETRO.TIEMPO_FINAL_DE_MEZCLA%TYPE;
BEGIN
  SELECT MARCA_DE_TIEMPO INTO v_MARCA_DE_TIEMPO FROM VARIABLE;
  SELECT TIEMPO INTO v_TIEMPO FROM VARIABLE;
  SELECT TIEMPO_DE_SIMULACION INTO v_TIEMPO_DE_SIMULACION FROM PARAMETRO;
  SELECT TIEMPO_MEZCLA INTO v_TIEMPO_MEZCLA FROM VARIABLE;
  SELECT TIEMPO_FINAL_DE_MEZCLA INTO v_TIEMPO_FINAL_DE_MEZCLA FROM PARAMETRO;

  IF v_MARCA_DE_TIEMPO = 'SIGA' THEN
    UPDATE VARIABLE SET TIEMPO = TIEMPO + 1;
    IF v_TIEMPO <= v_TIEMPO_DE_SIMULACION THEN
      IF v_TIEMPO_MEZCLA <= v_TIEMPO_FINAL_DE_MEZCLA THEN
        UPDATE VARIABLE SET TIEMPO_MEZCLA = v_TIEMPO_MEZCLA + 1;
      ELSE
        UPDATE VARIABLE SET TIEMPO_MEZCLA = 0;
      END IF;
    END IF;
  END IF;
END;

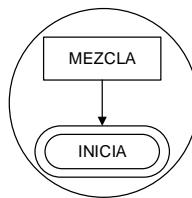
/
ALTER TRIGGER "TIEMPO_PASA" ENABLE
/

CREATE OR REPLACE EDITIONABLE TRIGGER "TIEMPO_PASA_1"
AFTER
  UPDATES OF "TIEMPO_MEZCLA"
ON "VARIABLE"
BEGIN
  UPDATE VARIABLE SET MARCA_DE_TIEMPO='PARE';
END;

/
ALTER TRIGGER "TIEMPO_PASA_1" ENABLE
/
    
```

Figura 5-14: Representación gráfica del evento TIEMPO PASA

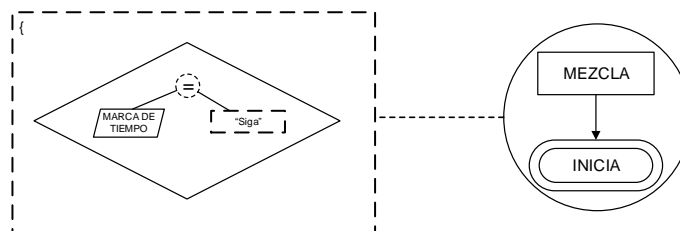
Fuente: Elaboración propia basada en Noreña *et al.* (2019)



CREATE OR REPLACE TRIGGER "MEZCLA_INICIA"

Figura 5-15: Condicional inicial del evento MEZCLA INICIA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)

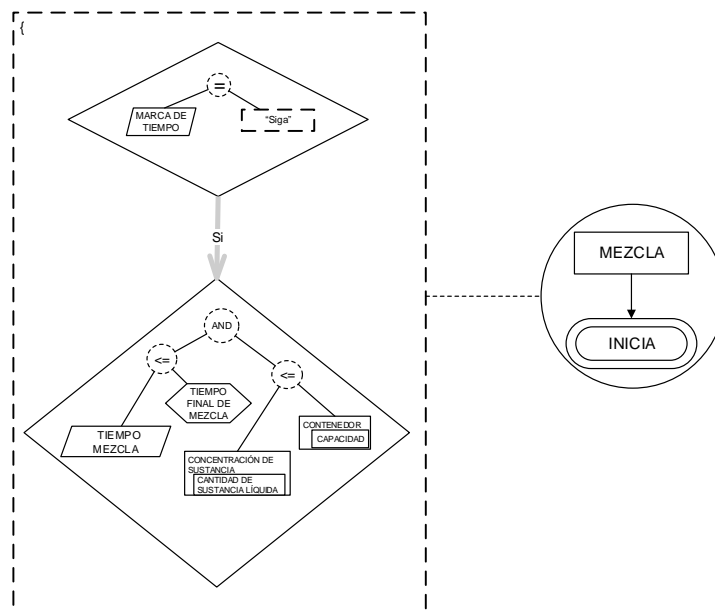


Sobre este condicional se aplica la regla #4 en la que se especifica el disparador del *trigger*. En la Figura 5-16 se muestra el condicional anidado con el condicional inicial.

```
CREATE TRIGGER "MEZCLA_INICIA"
  AFTER UPDATE OF "MARCA_DE_TIEMPO" ON "VARIABLE"
DECLARE
  v_MARCA_DE_TIEMPO VARIABLE.MARCA_DE_TIEMPO%TYPE;
BEGIN
  SELECT MARCA_DE_TIEMPO INTO v_MARCA_DE_TIEMPO FROM VARIABLE;
  IF v_MARCA_DE_TIEMPO = 'SIGA' THEN
    END IF;
END;
```

Figura 5-16: Condicional anidado con el condicional inicial del evento MEZCLA INICIA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



Para la siguiente transformación se debe aplicar la regla #8, que se refiere al uso de condicionales anidados y al uso de conceptos de los esquemas preconceptuales. Los conceptos asociados con la variable se representan con la siguiente sintaxis:

```
v_CANTIDAD_DE_SUSTANCIA_L
CONCENTRACION_DE_SUSTANCIA.CANTIDAD_DE_SUSTANCIA_LIQUIDA%TYPE;
```

Para la sentencia SELECT se tiene el siguiente código:

```
SELECT CANTIDAD_DE_SUSTANCIA_LIQUIDA INTO v_CANTIDAD_DE_SUSTANCIA_L FROM
CONCENTRACION_DE_SUSTANCIA ORDER BY CODIGO DESC FETCH FIRST 1 ROWS ONLY;
```

Al recuperar valores de tablas relacionadas con un concepto es necesario complementar la sintaxis de modo que se extraiga el último registro. Este proceso no es posible automatizarlo en las reglas, pues, dependiendo del problema que se vaya a resolver, se podría requerir también especificar una tupla en particular y no necesariamente la última. En este caso, se complementa la sintaxis con una instrucción para organizar los datos de la tabla y seleccionar lo necesario. Es de notar que cualquier tipo de sintaxis que se encargue de devolver el último registro es aplicable. Al final del registró, el *trigger* compila de manera satisfactoria y se incorpora en la estructura de la tabla VARIABLE, como se muestra en la Figura 5-17.

```
CREATE TRIGGER "MEZCLA_INICIA"
  AFTER UPDATE OF "MARCA_DE_TIEMPO" ON "VARIABLE"
DECLARE
  v_MARCA_DE_TIEMPO VARIABLE.MARCA_DE_TIEMPO%TYPE;

  v_TIEMPO_MEZCLA VARIABLE.TIEMPO_MEZCLA%TYPE;
  v_TIEMPO_FINAL_DE_MEZCLA PARAMETRO.TIEMPO_FINAL_DE_MEZCLA%TYPE;

  v_CANTIDAD_DE_SUSTANCIA_L
  CONCENTRACION_DE_SUSTANCIA.CANTIDAD_DE_SUSTANCIA_LIQUIDA%TYPE;
  v_CAPACIDAD CONTENEDOR.CAPACIDAD%TYPE;

BEGIN
  SELECT MARCA_DE_TIEMPO INTO v_MARCA_DE_TIEMPO FROM VARIABLE;

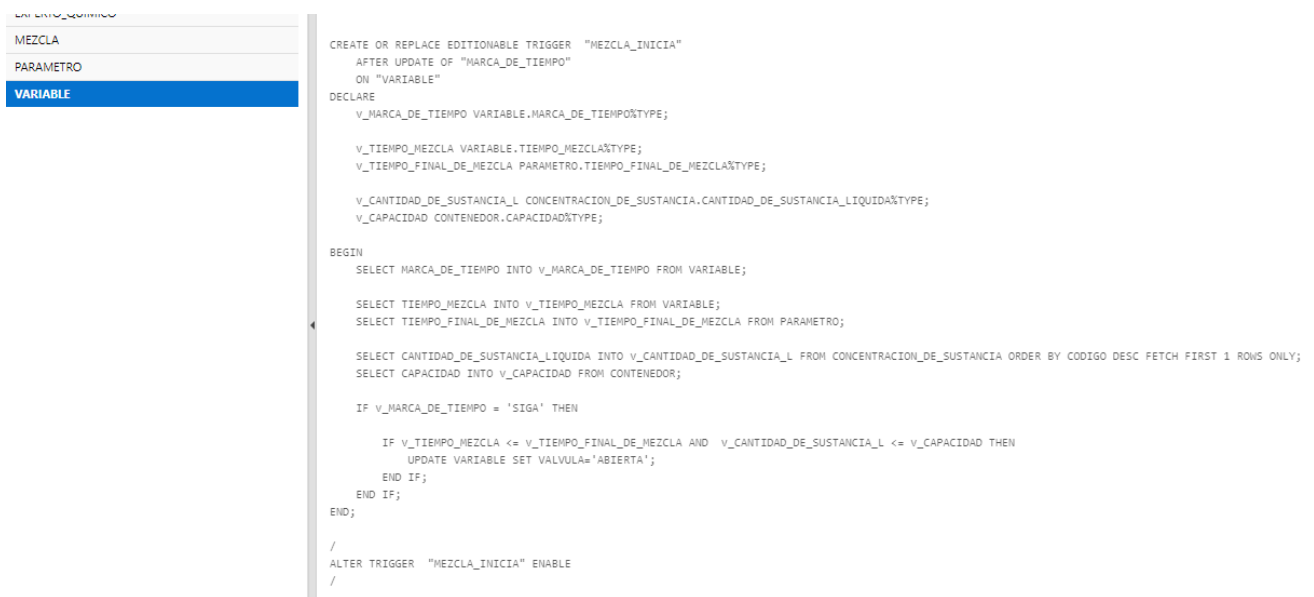
  SELECT TIEMPO_MEZCLA INTO v_TIEMPO_MEZCLA FROM VARIABLE;
  SELECT TIEMPO_FINAL_DE_MEZCLA INTO v_TIEMPO_FINAL_DE_MEZCLA FROM
PARAMETRO;

  SELECT CANTIDAD_DE_SUSTANCIA_LIQUIDA INTO v_CANTIDAD_DE_SUSTANCIA_L
FROM CONCENTRACION_DE_SUSTANCIA ORDER BY CODIGO DESC FETCH FIRST 1 ROWS
ONLY;
  SELECT CAPACIDAD INTO v_CAPACIDAD FROM CONTENEDOR;
```

```
IF v_MARCA_DE_TIEMPO = 'SIGA' THEN
  IF v_TIEMPO_MEZCLA <= v_TIEMPO_FINAL_DE_MEZCLA
  AND v_CANTIDAD_DE_SUSTANCIA_L <= v_CAPACIDAD THEN
    UPDATE VARIABLE SET VALVULA='ABIERTA';
  END IF;
END IF;
END;
```

Figura 5-17: Triggers finalmente incorporados en la tabla VARIABLE

Fuente: Elaboración propia



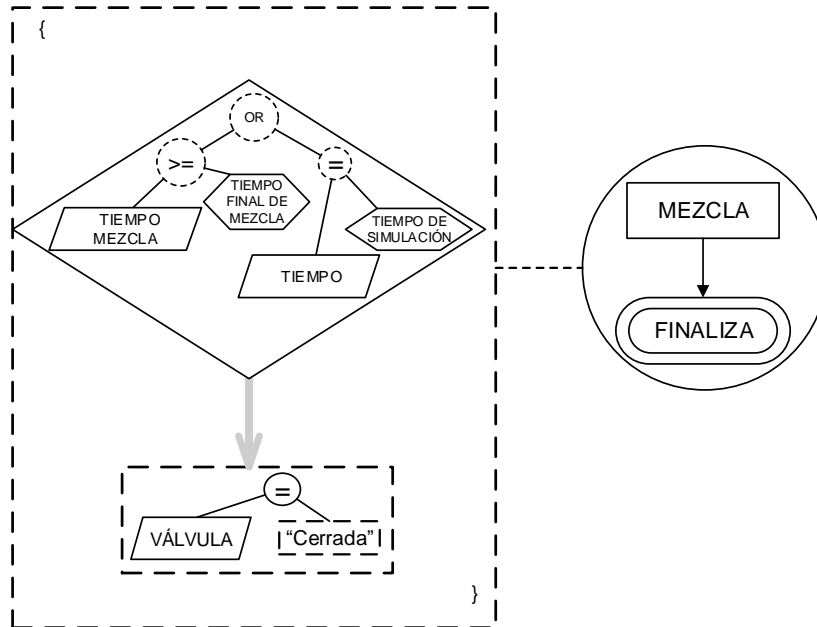
```
CREATE OR REPLACE EDITIONABLE TRIGGER "MEZCLA_INICIA"
AFTER UPDATE OF "MARCA_DE_TIEMPO"
ON "VARIABLE"
DECLARE
  v_MARCA_DE_TIEMPO VARIABLE.MARCA_DE_TIEMPO%TYPE;
  v_TIEMPO_MEZCLA VARIABLE.TIEMPO_MEZCLA%TYPE;
  v_TIEMPO_FINAL_DE_MEZCLA PARAMETRO.TIEMPO_FINAL_DE_MEZCLA%TYPE;
  v_CANTIDAD_DE_SUSTANCIA_L CONCENTRACION_DE_SUSTANCIA.CANTIDAD_DE_SUSTANCIA_LIQUIDA%TYPE;
  v_CAPACIDAD CONTENEDOR.CAPACIDAD%TYPE;
BEGIN
  SELECT MARCA_DE_TIEMPO INTO v_MARCA_DE_TIEMPO FROM VARIABLE;
  SELECT TIEMPO_MEZCLA INTO v_TIEMPO_MEZCLA FROM VARIABLE;
  SELECT TIEMPO_FINAL_DE_MEZCLA INTO v_TIEMPO_FINAL_DE_MEZCLA FROM PARAMETRO;
  SELECT CANTIDAD_DE_SUSTANCIA_LIQUIDA INTO v_CANTIDAD_DE_SUSTANCIA_L FROM CONCENTRACION_DE_SUSTANCIA ORDER BY CODIGO DESC FETCH FIRST 1 ROWS ONLY;
  SELECT CAPACIDAD INTO v_CAPACIDAD FROM CONTENEDOR;
  IF v_MARCA_DE_TIEMPO = 'SIGA' THEN
    IF v_TIEMPO_MEZCLA <= v_TIEMPO_FINAL_DE_MEZCLA AND v_CANTIDAD_DE_SUSTANCIA_L <= v_CAPACIDAD THEN
      UPDATE VARIABLE SET VALVULA='ABIERTA';
    END IF;
  END IF;
END;
/
ALTER TRIGGER "MEZCLA_INICIA" ENABLE
/
```

En la Figura 5-18 se muestra la representación gráfica completa del evento MEZCLA_FINALIZA. Inicialmente se analiza la estructura del evento para concluir que generación de código requiere la aplicación de la regla #9, la cual hace referencia a *triggers* que utilizan más de un disparador; en este caso, para MEZCLA_FINALIZA se utilizan las variables TIEMPO_MEZCLA y TIEMPO en su ejecución. Al manejar diferentes disparadores es necesario dividir el *trigger* en dos partes, una para cada disparador. Cualquier condición que se le suma al condicional inicial requiere su propio *trigger*. La notación establecida para llamar dichos *triggers* se basa en el nombre del *trigger* concatenado con la variable disparadora del mismo, en este caso el *trigger* se divide en MEZCLA_FINALIZA_TIEMPO_MEZCLA y MEZCLA_FINALIZA_TIEMPO. Sobre estos nombres también se aplica la restricción de los 30 *bytes*. La condición de ejecución de cada *trigger* es la condición del disparador original, como se aprecia en el código siguiente.

Al final de la ejecución, ambos *triggers* compilan correctamente y se incorporan también a la tabla VARIABLE, como se muestra en la Figura 5-19.

Figura 5-18: Notación completa del evento MEZCLA FINALIZA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



```

CREATE TRIGGER "MEZCLA_FINALIZA_TIEMPO_MEZCLA"
  AFTER UPDATE OF "TIEMPO_MEZCLA"
  ON "VARIABLE"
DECLARE
  v_TIEMPO_MEZCLA VARIABLE.TIEMPO_MEZCLA%TYPE;
  v_TIEMPO_FINAL_DE_MEZCLA PARAMETRO.TIEMPO_FINAL_DE_MEZCLA%TYPE;
BEGIN
  SELECT TIEMPO_MEZCLA INTO v_TIEMPO_MEZCLA FROM VARIABLE;
  SELECT TIEMPO_FINAL_DE_MEZCLA INTO v_TIEMPO_FINAL_DE_MEZCLA FROM
PARAMETRO;

  IF v_TIEMPO_MEZCLA >= v_TIEMPO_FINAL_DE_MEZCLA THEN
    UPDATE VARIABLE SET VALVULA='CERRADA';
  END IF;
END;

CREATE TRIGGER "MEZCLA_FINALIZA_TIEMPO"
  AFTER
  UPDATE OF "TIEMPO"
  ON "VARIABLE"

```

DECLARE

```
v_TIEMPO VARIABLE.TIEMPO%TYPE;  
v_TIEMPO_DE_SIMULACION PARAMETRO.TIEMPO_DE_SIMULACION%TYPE;
```

BEGIN

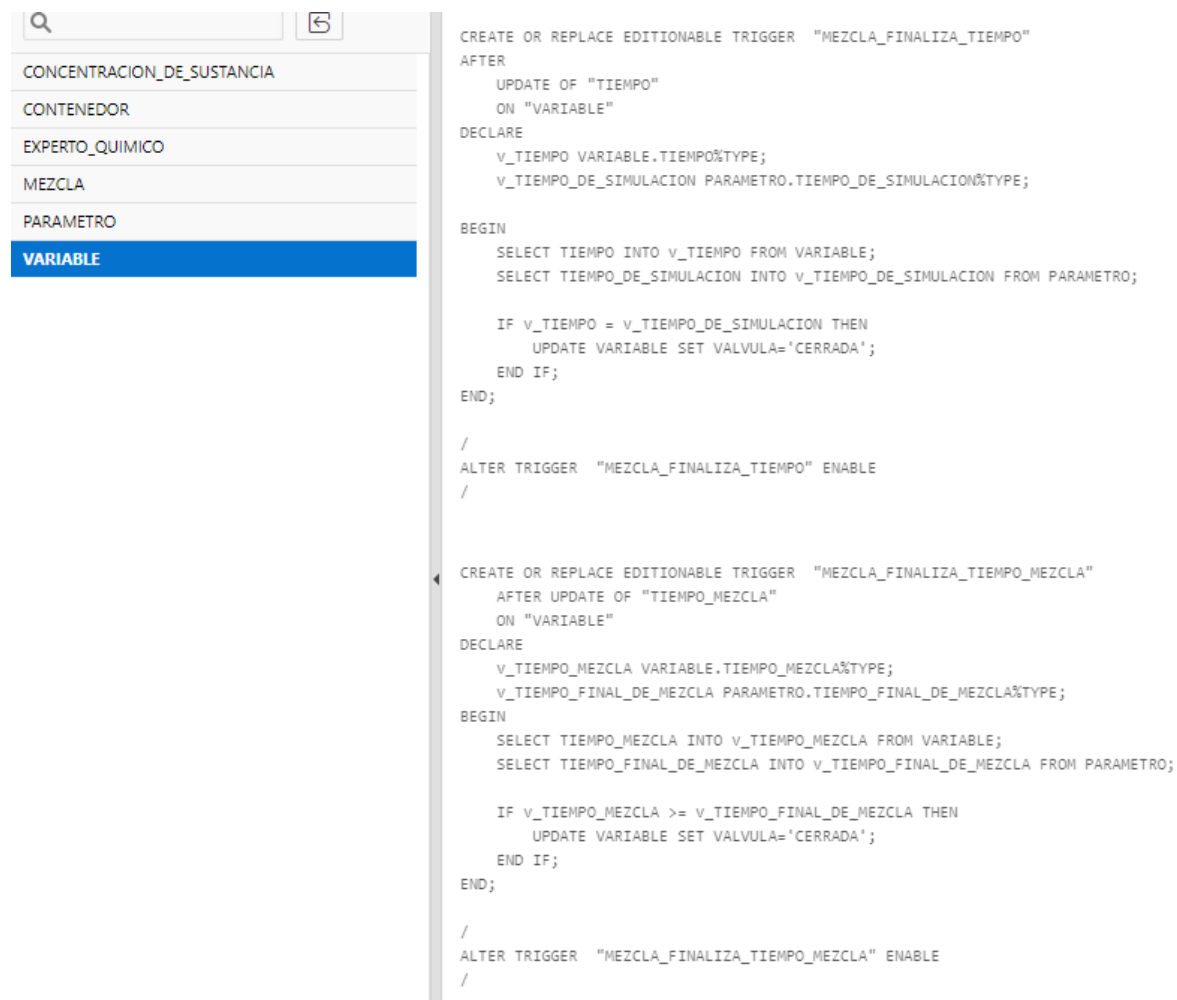
```
SELECT TIEMPO INTO v_TIEMPO FROM VARIABLE;  
SELECT TIEMPO_DE_SIMULACION INTO v_TIEMPO_DE_SIMULACION FROM PARAMETRO;
```

```
IF v_TIEMPO = v_TIEMPO_DE_SIMULACION THEN  
    UPDATE VARIABLE SET VALVULA='CERRADA';  
END IF;
```

END;

Figura 5-19: Otros *triggers* finalmente incorporados en la tabla VARIABLE

Fuente: Elaboración propia



```
CREATE OR REPLACE EDITIONABLE TRIGGER "MEZCLA_FINALIZA_TIEMPO"  
AFTER  
    UPDATE OF "TIEMPO"  
    ON "VARIABLE"  
DECLARE  
    v_TIEMPO VARIABLE.TIEMPO%TYPE;  
    v_TIEMPO_DE_SIMULACION PARAMETRO.TIEMPO_DE_SIMULACION%TYPE;  
BEGIN  
    SELECT TIEMPO INTO v_TIEMPO FROM VARIABLE;  
    SELECT TIEMPO_DE_SIMULACION INTO v_TIEMPO_DE_SIMULACION FROM PARAMETRO;  
  
    IF v_TIEMPO = v_TIEMPO_DE_SIMULACION THEN  
        UPDATE VARIABLE SET VALVULA='CERRADA';  
    END IF;  
END;  
  
/  
ALTER TRIGGER "MEZCLA_FINALIZA_TIEMPO" ENABLE  
/  
  
CREATE OR REPLACE EDITIONABLE TRIGGER "MEZCLA_FINALIZA_TIEMPO_MEZCLA"  
AFTER UPDATE OF "TIEMPO_MEZCLA"  
ON "VARIABLE"  
DECLARE  
    v_TIEMPO_MEZCLA VARIABLE.TIEMPO_MEZCLA%TYPE;  
    v_TIEMPO_FINAL_DE_MEZCLA PARAMETRO.TIEMPO_FINAL_DE_MEZCLA%TYPE;  
BEGIN  
    SELECT TIEMPO_MEZCLA INTO v_TIEMPO_MEZCLA FROM VARIABLE;  
    SELECT TIEMPO_FINAL_DE_MEZCLA INTO v_TIEMPO_FINAL_DE_MEZCLA FROM PARAMETRO;  
  
    IF v_TIEMPO_MEZCLA >= v_TIEMPO_FINAL_DE_MEZCLA THEN  
        UPDATE VARIABLE SET VALVULA='CERRADA';  
    END IF;  
END;  
  
/  
ALTER TRIGGER "MEZCLA_FINALIZA_TIEMPO_MEZCLA" ENABLE  
/
```

El último evento a transformar en código PL/SQL es CONCENTRACION DE SUSTANCIA INCREMENTA. Primero se debe aplicar la regla #12, que especifica las dependencias entre eventos. En este caso CONCENTRACION DE SUSTANCIA INCREMENTA posee dos dependencias MEZCLA INICIA y TIEMPO PASA, como se muestra en la Figura 5-20. Cuando esto ocurre, la regla indica que el condicional que dispara el *trigger* debe contener la última acción que se realiza en cada evento; sin embargo, la condición que dispara el evento es la actualización de cualquiera de estos dos elementos. En el caso de TIEMPO_PASA su última acción corresponde a una actualización sobre la variable TIEMPO_DE_MEZCLA, como se muestra en la Figura 5-21. En el caso de MEZCLA_INICIA su última acción corresponde a una actualización sobre la variable VALVULA, como se muestra en la Figura 5-22. El condicional de disparo de CONCENTRACIÓN DE SUSTANCIA INCREMENTA se puede apreciar en la Figura 5-23.

Figura 5-20: Dependencias entre eventos del caso de estudio

Fuente: Elaboración propia basada en Noreña *et al.* (2019)

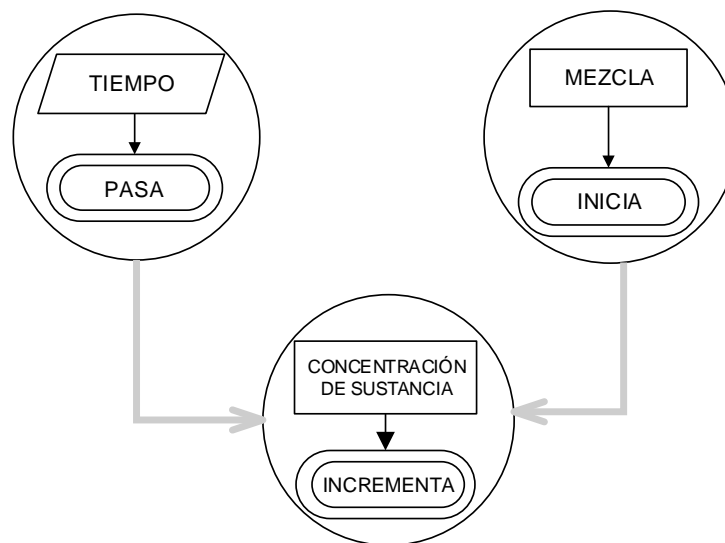


Figura 5-21: Restricción asociada con la versión final de TIEMPO PASA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)

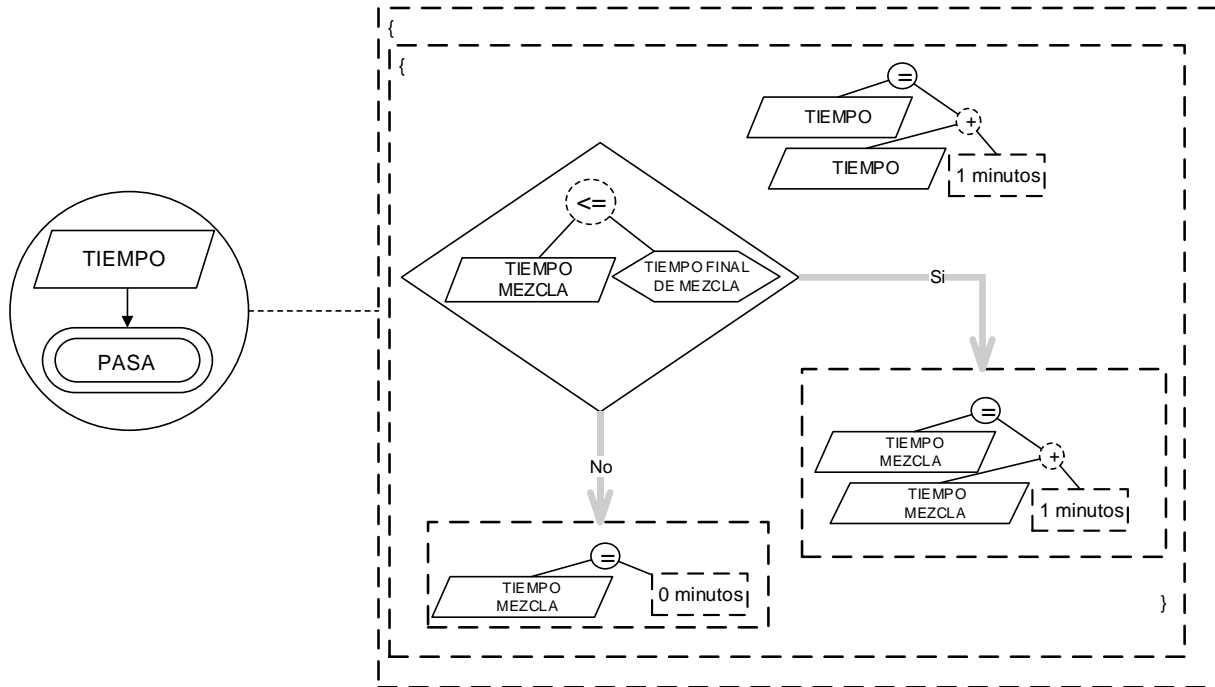


Figura 5-22: Restricción asociada con la versión final de MEZCLA INICIA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)

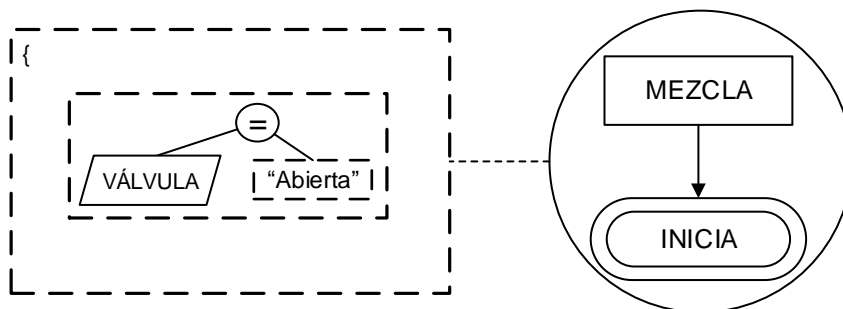
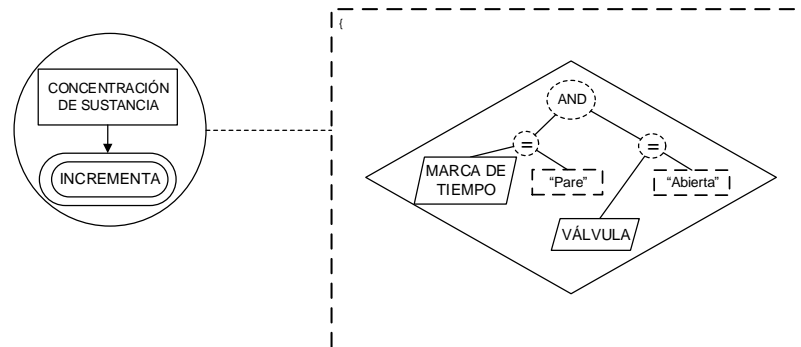


Figura 5-23: Condicional de disparo de CONCENTRACIÓN DE SUSTANCIA INCREMENTA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



El código correspondiente a ese condicional es el siguiente:

```
CREATE OR REPLACE EDITIONABLE TRIGGER "CONCENTRACION_SUSTANCIA_IN"
AFTER UPDATE OF "VALVULA" UPDATE OF "MARCA_DE_TIEMPO" ON "VARIABLE"
DECLARE
    v_MARCA_DE_TIEMPO VARIABLE.MARCA_DE_TIEMPO%TYPE;
    v_VALVULA VARIABLE.VALVULA%TYPE;

BEGIN
    SELECT MARCA_DE_TIEMPO INTO v_MARCA_DE_TIEMPO FROM VARIABLE;
    SELECT VALVULA INTO v_VALVULA FROM VARIABLE;

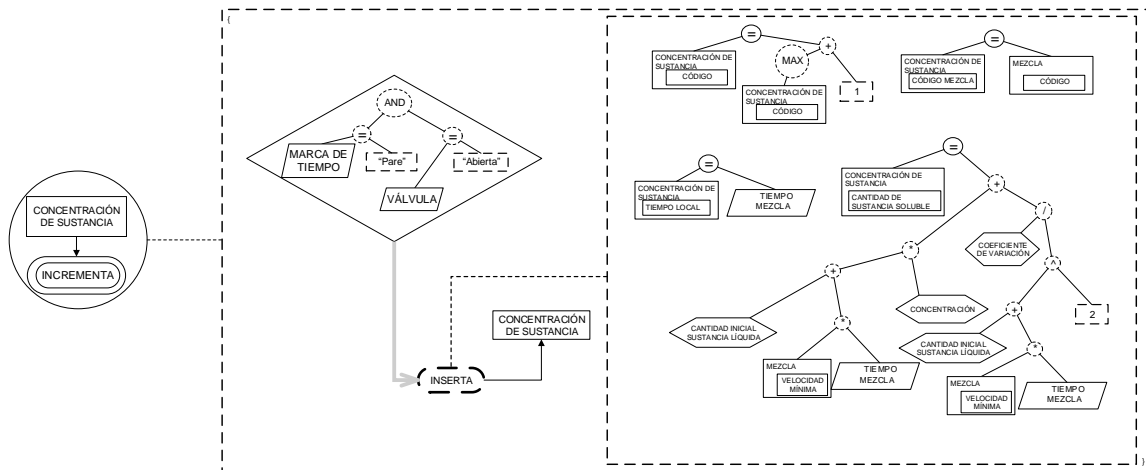
    IF v_MARCA_DE_TIEMPO = 'PARE' AND v_VALVULA = 'ABIERTA' THEN
        END IF;

END;
```

Luego, se establece que la relación dinámica atómica INSERTA dentro de la restricción (véase la Figura 5-24) representa un INSERT del concepto CONCENTRACION DE SUSTANCIA. Por lo tanto, después de declarar las variables necesarias de acuerdo con los datos, se realiza un INSERT con los datos solicitados.

Figura 5-24: Relación dinámica atómica INSERTA de CONCENTRACIÓN DE SUSTANCIA INCREMENTA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



```

CREATE TRIGGER "CONCENTRACION_DE_SUSTANCIA_I"
  AFTER UPDATE OF "MARCA_DE_TIEMPO" OR UPDATE OF "VALVULA" ON "VARIABLE"
DECLARE
  v_MARCA_DE_TIEMPO VARIABLE.MARCA_DE_TIEMPO%TYPE;
  v_VALVULA VARIABLE.VALVULA%TYPE;

  v_CODIGO_CONCENTRACION_DE CONCENTRACION_DE_SUSTANCIA.CODIGO%TYPE;
  v_CODIGO_MEZCLA_CONCENTRA
CONCENTRACION_DE_SUSTANCIA.CODIGO_MEZCLA%TYPE;
  v_CODIGO_MEZCLA MEZCLA.CODIGO%TYPE;

  v_TIEMPO_LOCAL_CONCENTRAC CONCENTRACION_DE_SUSTANCIA.TIEMPO_LOCAL%TYPE;
  v_TIEMPO_MEZCLA VARIABLE.TIEMPO_MEZCLA%TYPE;

  v_CANTIDAD_DE_SUSTANCIA_S
CONCENTRACION_DE_SUSTANCIA.CANTIDAD_DE_SUSTANCIA_SOLUBLE%TYPE;

  V_CANTIDAD_INICIAL_SUSTANCIA_L
PARAMETRO.CANTIDAD_INICIAL_SUSTANCIA_LIQ%TYPE;
  v_VELOCIDAD_MINIMA_MEZCLA MEZCLA.VELOCIDAD_MINIMA%TYPE;

  v_CONCENTRACION PARAMETRO.CONCENTRACION%TYPE;
  v_COEFICIENTE_DE_VARIACIO PARAMETRO.COEFICIENTE_DE_VARIACION%TYPE;

```

```
v_CANTIDAD_DE_SUSTANCIA_LIQUID
CONCENTRACION_DE_SUSTANCIA.CANTIDAD_DE_SUSTANCIA_LIQUIDA%TYPE;
v_CANTIDAD_MINIMA_SUSTANCIA_LI
CONCENTRACION_DE_SUSTANCIA.CANTIDAD_MINIMA_DE_SUSTANCIA_L%TYPE;

BEGIN
  SELECT MARCA_DE_TIEMPO INTO v_MARCA_DE_TIEMPO FROM VARIABLE;
  SELECT VALVULA INTO v_VALVULA FROM VARIABLE;

  SELECT CODIGO INTO v_CODIGO_CONCENTRACION_DE FROM
CONCENTRACION_DE_SUSTANCIA ORDER BY CODIGO DESC FETCH FIRST 1 ROWS ONLY;
  SELECT CODIGO_MEZCLA INTO v_CODIGO_MEZCLA_CONCENTRA FROM
CONCENTRACION_DE_SUSTANCIA;
  SELECT CODIGO INTO v_CODIGO_MEZCLA FROM MEZCLA;

  SELECT TIEMPO_LOCAL INTO v_TIEMPO_LOCAL_CONCENTRAC FROM
CONCENTRACION_DE_SUSTANCIA;
  SELECT TIEMPO_MEZCLA INTO v_TIEMPO_MEZCLA FROM VARIABLE;
  SELECT CANTIDAD_DE_SUSTANCIA_SOLUBLE INTO v_CANTIDAD_DE_SUSTANCIA_S FROM
CONCENTRACION_DE_SUSTANCIA;

  SELECT CANTIDAD_INICIAL_SUSTANCIA_LIQ INTO v_CANTIDAD_INICIAL_SUSTANCIA_L
FROM PARAMETRO;
  SELECT VELOCIDAD_MINIMA INTO v_VELOCIDAD_MINIMA_MEZCLA FROM MEZCLA WHERE
CODIGO = v_CODIGO_MEZCLA;

  SELECT CONCENTRACION INTO v_CONCENTRACION FROM PARAMETRO;
  SELECT COEFICIENTE_DE_VARIACION INTO v_COEFICIENTE_DE_VARIACIO FROM
PARAMETRO;

  SELECT CANTIDAD_DE_SUSTANCIA_LIQUIDA INTO v_CANTIDAD_DE_SUSTANCIA_LIQUID
FROM CONCENTRACION_DE_SUSTANCIA WHERE CODIGO = v_CODIGO_CONCENTRACION_DE;

  SELECT CANTIDAD_MINIMA_DE_SUSTANCIA_L INTO v_CANTIDAD_MINIMA_SUSTANCIA_LI
FROM CONCENTRACION_DE_SUSTANCIA WHERE CODIGO = v_CODIGO_CONCENTRACION_DE;

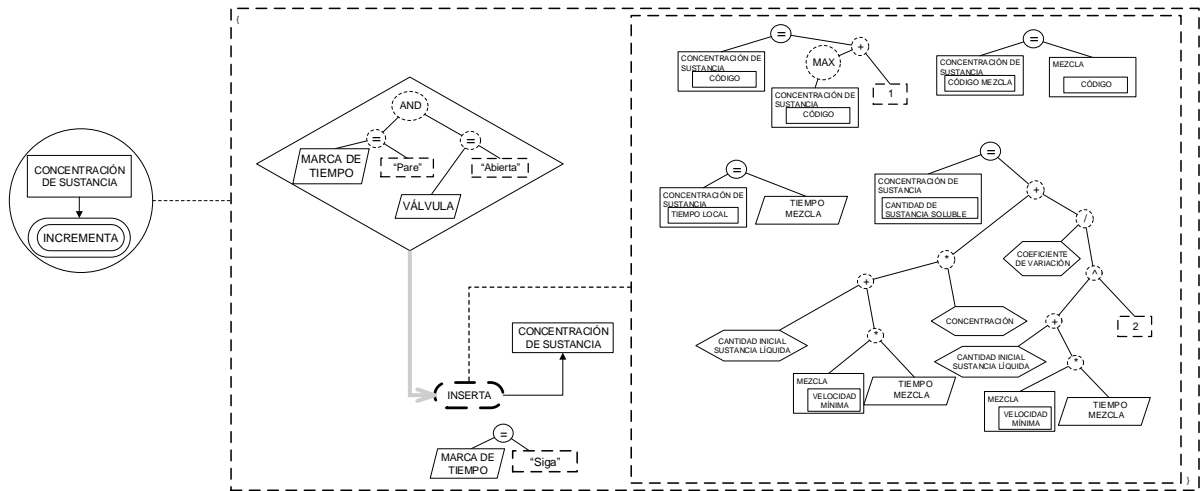
  IF v_MARCA_DE_TIEMPO = 'PARE' AND v_VALVULA = 'ABIERTA' THEN
    v_CODIGO_CONCENTRACION_DE := v_CODIGO_CONCENTRACION_DE + 1;
    v_CODIGO_MEZCLA_CONCENTRA := v_CODIGO_MEZCLA;
    v_TIEMPO_LOCAL_CONCENTRAC := v_TIEMPO_MEZCLA;
    v_CANTIDAD_DE_SUSTANCIA_LIQUID := v_CANTIDAD_INICIAL_SUSTANCIA_L +
v_CANTIDAD_MINIMA_SUSTANCIA_LI;
```

```
v_CANTIDAD_MINIMA_SUSTANCIA_LI := v_VELOCIDAD_MINIMA_MEZCLA *  
v_TIEMPO_MEZCLA;  
  
v_CANTIDAD_DE_SUSTANCIA_S := ((v_CANTIDAD_INICIAL_SUSTANCIA_L +  
(v_VELOCIDAD_MINIMA_MEZCLA * v_TIEMPO_MEZCLA)) * v_CONCENTRACION) +  
(power((v_CANTIDAD_INICIAL_SUSTANCIA_L + (v_VELOCIDAD_MINIMA_MEZCLA *  
v_TIEMPO_MEZCLA)), 2) / v_COEFICIENTE_DE_VARIACIO);  
  
INSERT INTO CONCENTRACION_DE_SUSTANCIA (CODIGO,  
CANTIDAD_DE_SUSTANCIA_LIQUIDA, CANTIDAD_MINIMA_DE_SUSTANCIA_L,  
CANTIDAD_DE_SUSTANCIA_SOLUBLE, CODIGO_MEZCLA, TIEMPO_LOCAL)  
VALUES (v_CODIGO_CONCENTRACION_DE, v_CANTIDAD_DE_SUSTANCIA_LIQUID,  
v_CANTIDAD_MINIMA_SUSTANCIA_LI, v_CANTIDAD_DE_SUSTANCIA_S, v_CODIGO_MEZCLA,  
v_TIEMPO_LOCAL_CONCENTRAC);  
END IF;  
END;
```

Además, el evento posee una asignación a la variable MARCA DE TIEMPO, que se encuentra entre las condiciones del disparador del *trigger* (véase la Figura 5-25). Por lo tanto, se requiere la aplicación de la regla #7 que especifica la separación del *trigger* para evitar el fenómeno de tabla mutante. El nuevo *trigger* tiene como condición disparadora la última acción que se realiza en el *trigger* que tiene el INSERT de CONCENTRACIÓN DE SUSTANCIA.

Figura 5-25: Condicional de disparo de CONCENTRACIÓN DE SUSTANCIA INCREMENTA

Fuente: Elaboración propia basada en Noreña *et al.* (2019)



```
CREATE TRIGGER "CONCENTRACION_DE_SUSTANCIA_I_1"
AFTER
    INSERT on "CONCENTRACION_DE_SUSTANCIA"
BEGIN
    UPDATE VARIABLE SET MARCA_DE_TIEMPO='SIGA';
END;
```

Una vez los dos *triggers* finales de CONCENTRACIÓN DE SUSTANCIA compilan, ya es posible iniciar la simulación. Para ello, se debe ejecutar manualmente la tríada dinámica EXPERTO QUÍMICO REGISTRA MEZCLA, en la cual se ingresa el código de la mezcla, se selecciona el contenedor y las sustancias líquida y soluble de la mezcla, se inserta la velocidad de entrada de la sustancia líquida y la velocidad de salida de sustancia líquida, se calcula la velocidad de entrada de la sustancia soluble, así como la velocidad de salida de sustancia líquida y la velocidad mínima de la mezcla, con las fórmulas que aparecen en la especificación de la relación dinámica. El alcance de esta Tesis de Maestría se restringe a los eventos del esquema preconceptual, y por ello esta relación dinámica se ingresa por consola o con una instrucción INSERT. En condiciones normales, esa relación dinámica debe corresponder a una interfaz gráfica de usuario en la que el experto químico puede ingresar la información. Esa primera información se refleja en el primer registro de CONCENTRACIÓN DE SUSTANCIA, que se aprecia bajo el código 500 en la Figura 5-26.

La relación dinámica culmina con una asignación que desencadena todo el proceso y activa la ejecución de los diferentes *triggers* programados con las reglas heurísticas en este caso de estudio. Así, la asignación del valor SIGA al campo MARCA_DE_TIEMPO de la tabla de VARIABLES, que permite la activación de los eventos se ingresa por consola de la siguiente manera:

```
UPDATE VARIABLE SET MARCA_DE_TIEMPO='SIGA';
```

Una vez se ingresa ese valor, los *trigger* corren de forma autónoma y se generan los registros adicionales en la tabla CONCENTRACION_DE_SUSTANCIA, que es la tabla donde se realiza la inserción de datos a partir del *trigger* CONCENTRACION_DE_SUSTANCIA_I, se generan las concentraciones 501 a 511 de la Figura 5-26, lo que valida el resultado del caso de estudio, con todos los *triggers* bien compilados y corriendo de forma autónoma para obtener los resultados del modelo. Con las condiciones iniciales que plantean Noreña *et al.* (2019), la cantidad de registros en la tabla debería ser superior a los 11 registros de este resultado, pero, debido a las limitaciones del entorno (que es una simulación con una cuenta universitaria, sin toda la capacidad de almacenamiento), hubo necesidad de restringir el registro en la tabla a esos 11 registros.

Figura 5-26: Contenido de la tabla CONCENTRACIÓN DE SUSTANCIA una vez se ejecuta la relación dinámica y se activan los *triggers* del caso de estudio

Fuente: Elaboración propia

EDIT	CODIGO	CANTIDAD_DE_SUSTANCIA_LIQUIDA	CANTIDAD_MINIMA_DE_SUSTANCIA_L	CANTIDAD_DE_SUSTANCIA_SOLUBLE	CODIGO_MIEZCLA	TIEMPO
	500	200	0	50	1	0
	501	200	1	66.9387260758403272134885395415816632665	1	1
	502	201	2	67.2714515166206648265930637225489019561	1	2
	503	202	3	67.6041740273610944437775111004440177907	1	3
	504	203	4	67.93689350294011760470418816752670106804	1	4
	505	204	5	68.26960997839913596543861754470178807152	1	5
	506	205	6	68.60232345373814952598103904156966278651	1	6
	507	206	7	68.93503982895715828633145325813032521301	1	7
	508	207	8	69.2677414040561622404898959438377535101	1	8
	509	208	9	69.6004487903516140645625825033001320053	1	9
	510	209	10	69.93314735389415576623064922594903876155	1	10
	511	210	11	70.26584828663314532581303252130085203408	1	11

6. Conclusiones y trabajo futuro

Conclusiones

En esta Tesis de Maestría se propuso un conjunto de reglas heurísticas que permiten la generación semiautomática de código PL/SQL tomando como base las representaciones de los eventos que se incluyen como característica de los esquemas preconceptuales. Se seleccionaron estos esquemas como punto de partida debido a su cercanía con el lenguaje natural del interesado y a la incorporación de nuevas características para especificar las relaciones dinámicas y los eventos, además del manejo de variables y parámetros, que le suministran riqueza expresiva y posibilidades de traducción hacia cualquier lenguaje de programación. Las reglas se validaron con un caso de estudio tomado de la literatura especializada y relacionado con la evolución en el tiempo de una mezcla en un contenedor en el cual se encuentran una sustancia líquida y una soluble. Partiendo de la representación del proceso en el esquema preconceptual, que incluyó la especificación de cada uno de los eventos y de la relación dinámica involucrada, se aplicaron manualmente las reglas para generar el código PL/SQL en el entorno *Oracle® Application Express*, que permite la simulación en línea en un espacio de trabajo libre con la funcionalidad plena del entorno real de Oracle®, aunque con algunas limitaciones de espacio. Se aprovecharon también las reglas heurísticas de Chaverra (2011) para construir las sentencias DDL de la base de datos y se crearon las sentencias DML para incorporar las condiciones iniciales y simular la interacción del usuario (denominado experto químico en el esquema preconceptual de partida).

Con las reglas heurísticas definidas en esta Tesis de Maestría fue posible la generación de la totalidad de los *triggers* en PL/SQL que posibilitaron una simulación completa del caso de estudio. Sin embargo, las restricciones en el almacenamiento que se puede hacer en el entorno de Oracle® impidieron que se pudieran generar todos los registros que se plantean en el caso de estudio. No obstante, se pudieron validar todos los *triggers*, pues se hicieron suficientes registros como para simular el comportamiento esperado de la mezcla en el contenedor. La sintaxis de la especificación de los eventos en el esquema preconceptual fue muy clara para la aplicación de las reglas y a partir de la interacción inicial del usuario, el proceso se controló mediante los *triggers* sin ninguna mediación de parte de usuario. Ahora, debido a los problemas de persistencia de las variables globales en Oracle®, se tomó la decisión de generar una tabla para las variables y una tabla para los parámetros. Con la persistencia de estos elementos ya definida, la simulación se pudo ejecutar sin inconvenientes.

Un obstáculo que se debió superar para la definición de las reglas heurísticas tiene que ver con el fenómeno conocido como “tabla mutante”, en el cual un *trigger* se dispara para una tabla al mismo tiempo que al final del mismo se trata de actualizar la tabla, generando un ciclo infinito que afecta la lógica del *trigger* y genera un error en la aplicación. La solución a este fenómeno consistió en dividir los eventos en los que se detectara en dos eventos con una sintaxis similar pero sin afectar simultáneamente el llamado y la actualización. Si bien se definieron las reglas que dan cuenta del fenómeno, también es posible alertar a los diseñadores de eventos para que, cuando usen el esquema preconceptual en la especificación de dichos eventos, tengan en consideración este fenómeno y aprendan del manejo de las reglas heurísticas de esta Tesis de Maestría sobre la forma de solucionarlo para no disparar errores en la aplicación.

Trabajo futuro

A partir de las reglas heurísticas definidas en esta Tesis de Maestría, se desprenden varias posibles líneas de trabajo futuro, a saber:

- Automatización completa de las reglas heurísticas en un prototipo que vincule ya sea un editor gráfico de esquemas preconceptuales (por ejemplo, actualmente se cuenta con plantillas de dichos esquemas en Visio™ y en Draw.io™ que podrían servir de base para la realización del trabajo, pues ambos entornos admiten procedimientos de manipulación de los elementos) o la sintaxis del UN-Lencep (Universidad Nacional del Colombia—Lenguaje para la especificación de los esquemas preconceptuales, en cuyo caso habría que considerar la construcción de un compilador de este lenguaje). La automatización es completamente posible y las reglas definidas en esta Tesis de Maestría dan cuenta de la totalidad del código que se presenta en el caso de estudio.
- Definición de una estrategia para superar las limitaciones asociadas con las variables globales cuando se trabaja con *triggers*. Si bien esta limitación se superó en esta Tesis de Maestría con la simulación de las tablas variable y parámetro, se debe estudiar con cuidado el fenómeno, dado que las simulaciones siempre tendrán presentes estos elementos.
- Análisis de los fenómenos temporales que afectan los *triggers*, por ejemplo tomando en consideración el trabajo de Behrend *et al.* (2009) y retomando la definición de su sintaxis para el PL/SQL que incorpora manejo temporal. Esta sintaxis posibilitaría la activación de los *triggers* aún en condiciones en que la base de datos se encuentre fuera de servicio.

Referencias

Breu, R., Agreiter, B., Farwick, M., Felderer, M., Hafner, M. y Innerhofer-Oberperfler, F. (2011). International journal of software and informatics, Vol. 5, No. 1-2, 267–290.

Behrend, A., Dorau, Ch. & Manthey, R. (2009). SQL Triggers Reacting on Time Events: An Extension Proposal. Lecture notes in Computer Sciences, Vol. 5739, 179–193.

Celko, J. (2011). Joe Celko's SQL for Smarties, 4th edition. Morgan Kaufmann: San Francisco.

Chaverra, J. J. (2011). Generación Automática de Prototipos Funcionales a Partir de Esquemas Preconceptuales. M.Sc. Thesis. Universidad Nacional de Colombia: Medellín.

Chochlik, M., Kostolny, J. & Martincova, P. (2015). Metamodel describing a relational database schema. Central European Researchers Journal, Vol.1, No. 1, 94–102

Egea, M. & Nadia, C. (2017). SQL-PL4OCL : An automatic code generator from OCL to SQL Procedural Language. In: Proceedings of the 2017 ACM/IEEE 20th International Conference on Model Driven Engineering Languages and Systems, Austin, USA.

Feuerstein, S. (2008). Oracle PL/SQL best practices. O'Reilly Media: Sebastopol.

Feuerstein, S. (2012). Oracle PL/SQL Programming. O'Reilly Media: Sebastopol.

Habringer, M., Moser, M. & Pichler, J. (2014). Reverse Engineering PL/SQL Legacy Code: An Experience Report. In: Proceedings of the 2014 IEEE International Conference on Software Maintenance and Evolution, Victoria, Canada.

Methakullawat , N., & Limpiyakorn, Y. (2014). Reengineering Legacy Code with Model Transformation. International Journal of Software Engineering and Its Applications. Vol. 8, No. 3, 97–110.

Noreña, P. A. (2013). Un mecanismo de consistencia en los eventos disparador y de resultado para los artefactos de UNC-Method. M.Sc. Thesis. Medellín: Universidad Nacional de Colombia.

Noreña, P. A., Zapata, C. M. & Villamizar, A. (2019). Representing Chemical Events by using Mathematical Notation from Pre-conceptual Schemas. IEEE Latin America, *en prensa*.

Rosenzweig, B. & Rakhimov, E. (2009). Oracle PL/SQL by example, fourth edition. Pearson Education Inc., Boston.

Serrano, D., Han, D. & Stroulia, E. (2015). From Relations to Multi-Dimensional Maps: Towards A SQL-to-HBase Transformation Methodology. In: Proceedings of the 2015 IEEE 8th International Conference on Cloud Computing, New York, USA.

Singh, S., Gupta, & Sabharwal. (2009). Automatic extraction of events from textual requirements specification. In: Proceedings of the Nature & Biologically Inspired Computing, NaBIC 2009 World Congress. Coimbatore, India.

Teorey, T., Lightstone, S., Nadeau, T. & Jagadish, H.V. (2005). Database Modeling and Design 4th Edition. Morgan Kaufmann: San Francisco.

Weinbach, & García. (2004). Una Extensión de la Programación en Lógica que incluye Eventos y Comunicación. VI Workshop de Investigadores en Ciencias de la Computación (WICC 2004), Buenos Aires, Argentina.

Zapata, C. M. The UNC-Method revisited: elements of the new approach. Lambert Academic Publishing, Saarbrücken.

Zapata Jaramillo, C. M., Chaverra Mojica, J. J., & Villa Buitrago, H. (2012). Un caso de estudio para la generación automática de código a partir de esquemas preconceptuales. Medellín: Cuaderno ACTIVA, Vol. 4, 9–24.

Zapata Jaramillo, C. M., Gelbukh, A., & Arango Isaza, F. (2006). Preconceptual Schema: A Conceptual-Graph-Like Knowledge Representation for Requirements Elicitation. Lecture Notes in Computer Science, Vol. 4293, 17–27.

Zapata, C. M., Noreña, P. A., & Vargas, F. (2014). The event interaction game: Understanding events in the software development context. Developments in Business Simulation and Experiential Learning, Vol. 41, 256–263.

Zapata, C. M., González, G., & Chaverra, J. J. (2011). Generación automática del diagrama entidad-relación y su representación en SQL desde un lenguaje controlado (UN-LENCEP). Revista Ingenierías Universidad de Medellín, Vol. 10, No. 18, 127–136.