Aplicación y Validación de Especificaciones Formales Ligeras en el Modelo Conceptual: Reduciendo la ambigüedad e incrementando la Conformidad entre los Requerimientos y el Código

Elizabeth Vidal-Duarte
Universidad La Salle Arequipa, Perú
evidal@ulasalle.edu.pe

Abstract— Una actividad importante en el análisis de requerimientos orientado a objetos es el identificar los conceptos relacionados a los requerimientos y crear un Modelo Conceptual del dominio. Muchas veces, dependiendo de la complejidad de los requerimientos a ser modelados no es posible que dicho modelo capture todos los detalles y restricciones existentes. Esto puede causar que el Modelo Conceptual sea sujeto de interpretaciones subjetivas que conlleven a errores de implementación mas adelante. Una de las formas de reducir este riesgo es mediante el uso de especificaciones formales ligeras como un complemento al Modelo Conceptual. La especificación debe expresar fehacientemente el comportamiento del sistema que está siendo modelado. Para ello las especificaciones serán validadas mediante casos de prueba refinados con análisis de cobertura. Habiendo validado en su totalidad la especificación, es posible generar un código más confiable que tenga una conformidad mucho más fuerte con respecto a los requerimientos Como ejemplo de aplicación se presenta la especificación formal ligera en VDM++ de un Electrocardiógrafo Digital. El ejemplo está basado en la descripción informal de las características de funcionamiento electrocardiógrafo, poniéndose especial atención en las características de la captura y almacenamiento de la señal.

Keywords-especificación formal, validación, análisis de cobertura, modelo conceptual, especificación formal ligera.

I. INTRODUCCION

Es ampliamente conocido que la parte más crítica del desarrollo de software corresponde a la identificación y especificación de requisitos [3,14]. Una actividad importante en el análisis de requerimientos es el identificar los conceptos relacionados a los requerimientos y crear un Modelo Conceptual del dominio. Un modelo conceptual ilustra los conceptos abstractos más significativos en el problema del dominio. La creación de conceptos es uno de los pasos esenciales en la orientación a objetos cuando investigamos el problema del dominio y poder construir software con posibilidad de reuso [23].

Para facilitar el proceso de desarrollo de software, muchos desarrolladores trabajan con una variedad de métodos y herramientas reconocidos, por ejemplo el Lenguaje de Modelado Unificado (UML) [13,17]. Aunque UML es el lenguaje estándar para modelamiento aún no está suficientemente refinado como para proveer toda la información relevante en una especificación [2]. Existe la necesidad de describir restricciones adicionales acerca de los objetos en el modelo. Dichas restricciones son descritas por lo general en lenguaje natural. La experiencia ha mostrado que esto siempre resulta en ambigüedades. Para evitar dichas ambigüedades se han desarrollado los llamados métodos formales.

Los Métodos Formales permiten una notación precisa, sin ambigüedades además de la aplicación de pruebas de correctitud. Sin embargo surge un enfoque más ligero que ha ganado popularidad. Hoy en día, el enfoque ligero de los métodos formales hace que los desarrolladores tomen ventaja de técnicas de especificación formal sin centrarse en las pruebas de correctitud [4]. En este artículo se hace uso de especificaciones formales ligeras.

Para incrementar la confianza en que la especificación formal refleja correctamente los requerimientos que fueron expresados, es necesario validar la especificación. Para ello es necesario utilizar técnicas de pruebas. El seleccionar casos de prueba para validar la especificación basándonos solamente en la técnica de caja negra [1], hace que exista la posibilidad de que no se ejecute el total de la especificación formal propuesta. Una técnica complementaria es el Análisis de Cobertura [20].

El principal aporte del presente trabajo es mostrar que la aplicación de especificaciones formales ligeras contribuye a incrementar la confiabilidad en la correctitud para la especificación de requerimientos. Así mismo se muestra como es posible refinar la validación de la especificación utilizando análisis de cobertura para luego generar código Java. Este código al ser generado directamente desde la especificación permite que tenga una conformidad mucho más fuerte con los requerimientos de usuario. Como caso de aplicación se presenta la especificación de un Electrocardiógrafo Digital. Si bien el resultado de este trabajo no nos proporciona una especificación completa, este trabajo brinda la descripción de las restricciones en la captura y almacenamiento de la señal que sirve como base para la ampliación del trabajo.

El resto del artículo está organizado de la siguiente manera: en la sección 2 se presenta los fundamentos en los que hemos basado nuestro trabajo: especificación formal ligera, pruebas, validación y análisis de cobertura. En la sección 3 se presentan las principales características de VDM++, la sintaxis que será utilizada en nuestro caso de estudio, y una breve descripción de la herramienta que será utilizada para validar nuestra propuesta. En la sección 4 se describe la metodología utilizada. En la sección 5 se presenta como caso de estudio un Electrocardiógrafo. Se describen los requisitos funcionales de manera informal para luego expresarlos de manera formal en VDM++. Además se presenta el análisis y validación del modelo propuesto utilizando la herramienta VDM++ToolBox y la generación de código Java. En la sección 6 se expondrán nuestras conclusiones.

II. MARCO TEORICO

A. Metodos Formales y Especificaciones Formales Ligeras

En [16] se definen Los Métodos Formales como técnicas basadas en matemáticas para la especificación, desarrollo y verificación de sistemas de hardware y software. Estas técnicas permiten determinar si las especificaciones son satisfechas, si la implementación de un sistema es correcta y permiten también realizar pruebas rigurosas a las propiedades del sistema. Los Métodos Formales se centran en una notación conocida como Lenguaje de Especificación Formal. Este lenguaje basado en matemáticas permite una notación precisa, sin ambigüedades. Además permite la consistencia y correctitud en las especificaciones a ser expresadas. La especificación formal de requerimientos y la verificación formal de software son pilares de los métodos formales. Sin embargo surge un enfoque más ligero que ha ganado popularidad: la Especificación Formal Ligera, la cual se centra más en la especificación que en las pruebas. Hoy en día, el enfoque ligero de los métodos formales hace que los desarrolladores tomen ventaja de técnicas de especificación formal sin centrarse en las pruebas de correctitud [4].

El enfoque ligero implica un lenguaje de especificación formal con semánticas formales. Aunque las pruebas formales están fuera del alcance del método formal ligero, la verificación formal es posible si esta fuera necesaria o requerida. Las ventajas de un método formal ligero se resumen a continuación: fácil de aprender, fácil de aplicar, descripción no ambigua: el lenguaje formal brinda una herramienta para especificar sin la ambigüedad introducida por las técnicas de descripción informales. Descripciones completas: en una especificación formal, se definen de manera obvia y precisa que tipo de propiedades son descritas y que se pospone para futuros refinamientos. [4,22].

Algunos métodos formales facilitan la especificación en un estilo explícito de manera que una herramienta pueda interpretar o generar código de este tipo de especificaciones. La ventaja de una especificación ejecutable es que pueden aplicarse técnicas de prueba (testing) para incrementar la confianza en la especificación [11].

B. Validación y Pruebas

Las especificaciones formales son realizadas generalmente para lograr una mejor comprensión del comportamiento deseado de un sistema, o para verificar que algún diseño tiene ciertas propiedades, como por ejemplo seguridad. Cualquiera que sea el propósito, que la especificación sea sintácticamente correcta y tenga los tipos correctos no es suficiente. La especificación además debe expresar fehacientemente el comportamiento del sistema que está siendo modelado [8].

La Validación es el proceso que incrementa la confianza en que la especificación formal refleja correctamente los requerimientos que fueron expresados de manera informal para el sistema que está siendo modelado. Para validar una especificación es necesario utilizar técnicas de pruebas (testing). La técnica utilizada en la primera parte de nuestra propuesta es la llamada Prueba de Caja Negra, técnica basada en la descripción de requerimientos [1].

C. Analisis de Cobertura

El seleccionar casos de prueba para validar la especificación basándonos solamente en la técnica de caja negra, hace que exista la posibilidad de que no se ejecute el total de la especificación formal propuesta. Una técnica complementaria es el Análisis de Cobertura.

El análisis de cobertura es la técnica que permite analizar y evaluar qué partes del código no fueron testeados. Permite encontrar fragmentos del programa que no son ejecutados por los casos de prueba. Ayuda a crear casos de prueba adicionales que incrementen la cobertura. Determinar un valor cuantitativo de la cobertura (que es, de manera indirecta, una medida de la calidad del programa). Adicionalmente, el análisis de cobertura también permite la identificación de casos de prueba redundantes, que no incrementan la cobertura [20,21].

Para nuestra propuesta utilizaremos el análisis de cobertura para analizar y evaluar que partes de la especificación formal no fueron ejecutadas y con eso poder crear nuevos casos de prueba. Esto se reflejará en una especificación formal refinada, más exacta y correcta.

III VIENNA DEVELOPMENT METHOD

En el presente trabajo, se decidió utilizar Vienna Development Method++ (VDM++) como lenguaje de especificación formal ligero, ya que éste permite la descripción y validación de modelos orientados a objetos [16]. Además, es ampliamente conocida la aplicación de VDM++ en la industria [19]. El lenguaje está basado en VDM-SL [15] el cual es un lenguaje de especificación formal estandarizado bajo la International Organization for Standarization (ISO). En esta sección se presenta la sintaxis de VDM++ relevantes para nuestro trabajo.

A. Definicion de Clases

De acuerdo a [16] los modelos en VDM++ consisten en un conjunto de clases. Una clase representa una colección de objetos que comparten elementos comunes como atributos u operaciones. La estructura de la descripción de una clase se muestra en la figura 1. La clase se representa con la palabra reservada `class`, seguida por el nombre de la clase. La descripción consta de varios bloques, precedidos por la palabra reservada que indica el tipo de elemento descrito en dicho bloque. En figura 1 podemos apreciar que una clase en VDM++ tiene los siguientes bloques: a) Variables de Instancia (`instance variables`), las cuales modelan el estado interno del objeto. b) Tipos (`types`): presenta la definición de tipos de datos. c) Valores (`values`): permite la definición de constantes. d) Operaciones (`operations`): define operaciones que pueden modificar las variables de instancia.

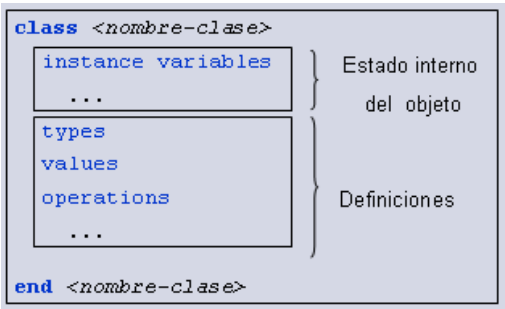

Figura. 1. Estructura de clase en VDM++

B. Tipos

En cuanto a los tipos de dato, VDM++ cuenta con tipos básicos y tipos compuestos. Entre los tipos básicos se tiene: boleanos (`bool`), naturales (`nat, nat1`), reales (`real`) y caracter (`char`). En los tipos compuestos presenta: conjuntos (`set of`), secuencias (`seq of`), mapeos (`map to`), entre otros. Cada uno de estos tipos tiene operaciones pre-definidas. Para la aplicación de nuestro trabajo queremos resaltar las operaciones pre-definidas para el tipo secuencia mostradas en la figura 2 [6,16].

Operador	Semántica	Tipo
hd l	primer elemento de la lista	seq1 of A -> A
tl l	cola de la lista	seq1 of A -> seq of A
len l	largo de lista	seq of A -> nat
inds l	índices de la lista	seq of A -> set of A
l(**i**)	elemento en la i-ésima posición	seq of A * nat1 -> A
l1 = l2	compara si dos listas son iguales	seq of A* seq of A ->bool

Figura. 2. Operaciones pre definidas sobre el tipo secuencia

C. Expresiones

Las expresiones son utilizadas para describir cálculos que no producen efectos secundarios, esto significa que nunca podrán afectar el valor de una variable de instancia (a menos que contenga un llamado a operación). Las expresiones pueden ser evaluadas mediante el remplazo del identificador utilizado en la expresión con valores actuales. La evaluación de una expresión da como resultado un solo valor. VDM++ tiene 25 categorías diferentes de expresiones [6]. Una de las categorías más utilizadas para la definición de pre-condiciones, post-condiciones en invariantes son las expresiones cuantificadoras.

D. Expresiones cuantificadoras

Las expresiones cuantificadoras son un tipo de expresión lógica. Son utilizadas de forma frecuente cuando es necesario realizar una aserción acerca de una colección de valores. Existen dos tipos de expresiones cuantificadoras: cuantificador universal (`forall`) y cuantificador existencial (`exists`). Ambas enlazan una o más variables a un tipo de VDM++ o, a un valor que pertenece a un conjunto, y los evalúa contra una expresión booleana [6,8,16].

E. Invariantes

Si las variables de instancia especificadas en una clase contienen valores que no deberían ser permitidos, entonces es posible restringir dichos valores por medio de invariantes. El resultado será que el tipo es restringido a un subconjunto de los valores originales [6,8]. Para especificar una invariante se utiliza la palabra reservada `inv` a continuación de la definición de todas las variables de instancia declaradas en la clase.

F. Operaciones

En VDM++ los algoritmos son definidos por operaciones. Las operaciones pueden manipular tanto variables globales y variables locales. Las operaciones pueden ser definidas de forma explícita (mediante un algoritmo explícito) o de forma implícita (mediante el uso de pre-condiciones y post-condiciones). Para poder ejecutar nuestras operaciones en el intérprete de la herramienta, las operaciones deben ser

definidas explícitamente. VDM++ también permite agregar pre-condiciones y post-condiciones en las operaciones explícitas [6,16]. La idea principal de las pre-condiciones y post-condiciones es que una clase y sus usuarios tienen un "contrato". El usuario debe garantizar ciertas condiciones antes del llamado a la operación (pre-condición) y por su parte la clase garantiza que ciertas propiedades serán verdaderas cuando termine la operación (post-condición) [10].

G. La Herramienta

VDM++ToolBox [5] es una herramienta integrada que soporta el análisis de modelos VDM++ vía análisis sintáctico, análisis de tipos, generación de condiciones de integridad y análisis de cobertura, además que permite la generación de código Java y C. La herramienta además posee un intérprete que permite validar las especificaciones. La herramienta también permite convertir directamente diagramas de clase UML creados en Rational Rose en especifacion VDM++.

Una vez realizada la especificación en VDM++, es necesario verificar si las clases concuerdan con las reglas sintácticas de VDM++. El analizador sintáctico verifica si la sintaxis de la especificación es correcta. El analizador sintáctico reporta los errores, si los hubiera, y realiza supuestos sobre cual pudo haber sido el error. Luego que la especificación ha pasado el analizador sintáctico, debe verificarse la correctitud de los tipos de acuerdo a las reglas de VDM++. Habiendo garantizado tanto la correctitud sintáctica como la correctitud de tipos, VDM++ToolBox provee soporte para validación de la especificación a través de la ejecución de pruebas utilizando un intérprete. El intérprete permite ejecutar partes de la especificación utilizando valores seleccionados por el desarrollador (casos de prueba).

IV METODOLOGIA

La metodología seguida para el desarrollo de nuestro trabajo se describe a continuación:

1. Captura de los requerimientos funcionales en lenguaje natural.
2. Creación del Modelo Conceptual
3. Especificación formal inicial de acuerdo a los requerimientos obtenidos.
4. Verificación de la sintaxis y la correctitud de los tipos de acuerdo a las reglas de VDM++.
5. Validación de la Especificación Formal mediante la ejecución de casos de prueba.
6. Análisis de Cobertura de la Especificación Formal.
7. Refinamiento de la validación de la especificación mediante la generación de nuevos casos de prueba.
8. Generación de Código Java.

En términos generales podemos mencionar que en los pasos 4 y 5 es posible especificar los requerimientos de forma clara y sin ambigüedad. Sin embargo son en los pasos del 6 al 7 en donde no solo validamos la especificación, sino garantizamos que tengamos tantos casos de prueba como sean necesarios para garantizar que toda la especificación ha sido validada. Los pasos 6 y 7 se realizan tantas veces sea necesario hasta lograr ejecutar y validar toda la especificación. Finalmente el paso 8 nos permitirá generar código Java previamente validado.

V APLICACIÓN: ELECTROCARDIOGRAFO

Para poder mostrar la propuesta que plantea nuestro artículo, hemos considerado el caso del funcionamiento de un electrocardiógrafo digital. El resultado de este trabajo no nos proporciona una especificación completa del electrocardiógrafo, sino brinda una descripción razonable de la especificación de una de las funcionalidades consideradas como críticas: el correcto almacenamiento de la señal electrocardiográfica.

A. Requerimientos Funcionales

Nuestro cuerpo está compuesto por millones de células que precisan de oxígeno y otras sustancias para su funcionamiento, las mismas que se encuentran en la sangre. Es el aparato circulatorio el encargado de realizar su distribución por todo el organismo. El corazón es el órgano fundamental de este sistema. Se encarga de dar el impulse necesario para que la sangre cumpla su recorrido. El corazón es un músculo que nunca descansa, pues de su permanente y correcto funcionamiento dependen nuestras vidas. La principal herramienta clínica que nos permite determinar el estado funcional del corazón es el electrocardiógrafo. El electrocardiógrafo genera un registro electrocardiográfico que proporciona información importante sobre la actividad eléctrica del corazón para determinar su estado funcional [12,18].

La señal electrocardiográfica es una señal de alto valor diagnóstico, para diversos tipos de enfermedades y patologías, directa o indirectamente relacionadas con el funcionamiento del corazón. El correcto funcionamiento de un electrocardiógrafo puede significar la diferencia entre la vida y la muerte (una lectura equivocada del electrocardiógrafo puede llevar a un diagnóstico equivocado). La correcta lectura y captura de la señal se convierten así en un componente crítico de su funcionamiento [9].

Para registrar un trazo electrocardiográfico, es necesaria la colocación de electrodos en partes específicas de la

superficie del cuerpo del paciente. La disposición específica que guardan los electrodos recibe el nombre de "derivación". Dependiendo de la colocación de los electrodos, se obtienen distintas derivaciones o pares de puntos. Las derivaciones que más se utilizan para hacer diagnóstico electrocardiográfico, se pueden clasificar en [18]:

- Bipolares. Las derivaciones estándares bipolares (llamadas D_I, D_{II} y D_{III}) se obtienen con el denominado triángulo de Einthoven, que se muestra en figura 3, para registrar los potenciales eléctricos en el plano frontal. Los electrodos se colocan en el brazo izquierdo (LA), el brazo derecho (RA), la pierna izquierda (LL) y la pierna derecha (RL), la que actúa como tierra. Debe hacerse un buen contacto con la piel con la ayuda de una pasta especial que se frota con la piel, y entonces colocar los electrodos en las posiciones mencionadas anteriormente.

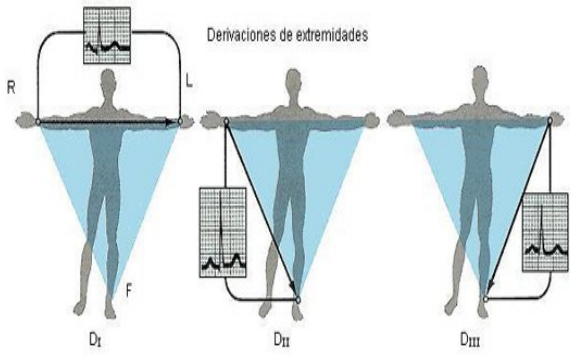

Figura 3: Derivaciones Bipolares

- Aumentadas. Estas representan la diferencia de potencial entre una de las extremidades y un electrodo correspondiente a la central terminal de Goldberg. Estas derivaciones se conocen como aVR, aVL y aVF y tienen una orientación diferente a las Bipolares, se aprecian en figura. 4.

- Precordiales Unipolares. Estas representan la diferencia de potencial entre electrodos colocados en partes específicas del tórax del paciente y un electrodo indiferente denominado central terminal de Wilson. Estas derivaciones se conocen como V_1, V_2, V_3, V_4, V_5 y V_6, se observan en figura 5.

  Como se puede apreciar, un electrocardiograma, para propósitos de diagnóstico, debe permitir el registro de 12 derivaciones

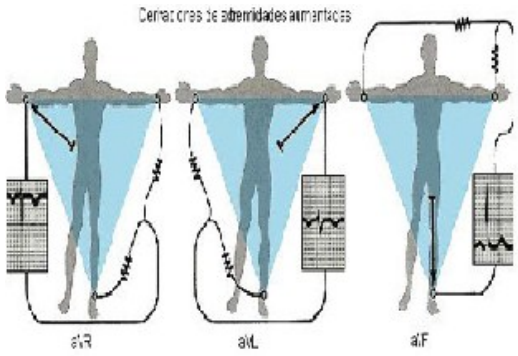

Figura 4: Derivaciones Aumentadas

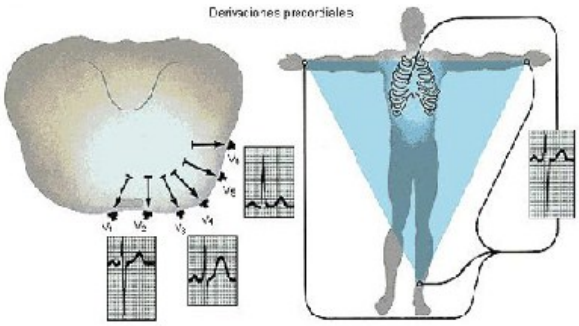

Figura 5:Derivaciones Extendidas

B. El Modelo

De acuerdo a [23], un Modelo Conceptual puede ser presentado por clases con sus respectivos atributos y operaciones. En el presente trabajo se ha utilizado UML [13,17] para modelar la clase que representa nuestro caso en estudio (tal como se muestra en la figura 6). En nuestro modelo solo se ha considerado una clase que representa el concepto más relevante con respecto a la captura de la señal del electrocardiograma: la clase Examen.

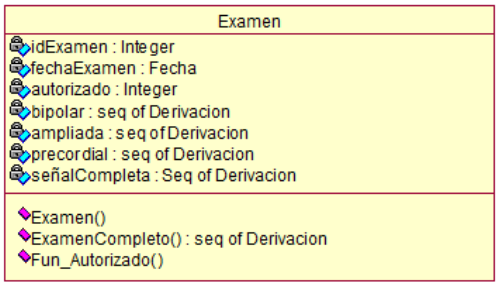

Figura 6: Modelo Conceptual

La clase presenta los atributos que representan el código del examen: `idExamen`, la fecha del examen: `fechaExamen`, la autorización del examen: autorizado, y las señales de la lectura del electrocardiógrafo: `bipolar, aumentada` y

precordial. También la clase presenta el Constructor y algunas operaciones de funcionamiento.
Como se puede apreciar en el Modelo Conceptual no es posible mostrar las restricciones referidas a las propiedades que garanticen la correctitud de la implementación de la captura de la señal.

C. *Especificación Formal Ligera*

Como parte esencial se hace necesario complementar el Modelo Conceptual con especificaciones formales ligeras que expresen las propiedades y restricciones que no son evidentes en el Modelo.

La especificación formal ligera del electrocardiógrafo fue realizada en VDM++ y se muestra en la figura 7. Podemos identificar como elemento más importante a la clase `Examen`, donde la información relevante está dada por el código del examen: `idExamen` (línea 8), a qué paciente se le realiza el examen: `idPaciente` (línea 9), `fechaExamen` (línea 10), la autorización para mostrar el examen: `autorizado` (línea 11) y las lecturas de señal que realiza el electrocardiógrafo (derivaciones): `bipolar, aumentada y precordial` (líneas 14, 17 y 20). Es importante resaltar que para que un examen sea considerado como satisfactorio deben realizarse los tres tipos de derivaciones.
Hemos identificado algunas consideraciones importantes que garanticen la correctitud en el almacenamiento de la señal electrocardiográfica y que no se aprecian en el Modelo Conceptual mostrado en la figura 7: a) cuando se realice la derivación bipolar debe garantizarse que se realizaron 3 lecturas, b) cuando se realice la derivación aumentada debe garantizarse que se realizaron 3 lecturas, c) cuando se realice la derivación precordial debe garantizarse que se realizaron 6 lecturas, d) es necesario que se realicen los tres tipos de derivaciones para que el examen sea considerado satisfactorio.

La primera consideración se refiere a la variable `bipolar`. La variable bipolar estará conformada por tres valores reales que representan las derivaciones D_I, D_{II} y D_{III}. Cada uno de los valores deberá ser diferente a 0 (que representaría la ausencia de señal). Estas restricciones son presentadas en forma de invariante, tal como se observa en la línea 15 de figura 7.

La segunda consideración se refiere a las restricciones de la variable `aumentada`. La variable aumentada está conformada también por tres valores reales que representan las derivaciones aVR, aVL y aVF. Cada uno de los valores deberá ser diferente a 0 (que representaría la ausencia de señal). Estas restricciones son presentadas en forma de invariante, tal como se observa en la línea 18 de figura 7

```
3  class Examen
4  types
5  public Fecha =nat1*nat1*nat1;
6  public Derivacion =real;
7  instance variables
8    public idExamen :nat1;
9    public idPaciente :nat1;
10   private fechaExamen : Fecha;
11   public autorizado : nat := 0;
12   inv autorizado>=0 and autorizado<=1;
13
14   private bipolar : seq of Derivacion:=[1, 1, 1];
15   inv bipolar <> [] and len bipolar =3 and (forall x in set inds bipolar & bipolar(x) <> 0);
16
17   private ampliada :seq of Derivacion:=[-1,-1,-1];
18   inv ampliaca <> [] and len ampliaca =3 and forall x in set inds ampliada & ampliada(x) <> 0;
19
20   private precordial : seq of Derivacion:= [-1,-1,-1,-1,-1,-1];
21   inv precordial <> [] and lenprecordial = 6 and (forall x in set inds precordial & precordial(x) <> 0);
26 operations
27 public Examen : nat1* nat1 * Fecha * seq of Derivacion * seq of Derivacion* seq of Derivacion *nat==> Examen
28   Examen(idEx,idPac,fec,bip, amp, prec, aut) ==
29     (idExamen:=idEx; idPaciente:=idPac; fechaExamen:=fec;
30      bipolar :=bip; ampliada :=amp; precordial:=prec; autorizado:=aut)
31   pre len bipolar = 3 and len ampliada = 3 and len precordial = 6;
32
33   public ExamenCompeto: () ==> seq of Derivacion
34   ExamenCompeto() ==
35         senalCompleta := (bipolar ^ ampliada ^ precordial);
36         return senalCompleta;)
37    pre len senalCompleta =12;
38
39    public Fun_Autorizado: ()==> seq of char *seq of Derivacion
40   Fun_Autorizado() ==
41   if autorizado = 1
42         then let siautorizado = ExamenCompleto()
43         in return mk_("Autorizado", siautorizado)
44    else let noautorizado= senalCompletanoaut
45         in return mk_("No_autorizado", noautorizado)
46
47 end Examen
```

Figura 7: Especificación Formal Ligera Electrocardiógrafo

La tercera consideración está referida a las restricciones de la variable precordial. La variable precordial estará conformada por seis valores reales que representan las derivaciones V_1, V_2, V_3, V_4, V_5 y V_6. Cada uno de los valores deberá ser diferente a 0 (que representaría la ausencia de señal). Estas restricciones son presentadas en forma de invariante, tal como se observa en la línea 21 de figura 7.
La cuarta consideración está dada a garantizar que se realizó un Examen satisfactorio a un Paciente. Esto sólo se logra cuando se realizaron las 12 derivaciones. Para esto se ha creado una nueva variable de instancia a la que hemos llamado senalCompleta (línea 23 en figura. 7). Esta variable contendrá la concatenación de las derivaciones bipolar, ampliada y precordial. La concatenación es realizada en el método `ExamenCompleto`. La propiedad que debe respetarse es que el tamaño de `senalCompleta` deberá ser igual a 12. La especificación explicada en este párrafo se presenta entre las líneas 33 al 37 de figura 7.

Además se ha especificado la operación `Fun_Autorizado` (líneas 39 al 45) y el constructor (líneas 27 al 31). En la operación `Fun_Autorizado` se ha puesto una condicional, la cual invoca a la operación

ExamenCompleto si es que la autorización es 1 como se describe en las líneas 41 a 43. De caso contrario no se mostrará la señal completa como se aprecia en las líneas 44 y 45 de la especificación. El empleo de la condicional nos ayuda a ver con mayor claridad el análisis de cobertura mostrándonos que líneas no se ejecutaron coloreándolas y el porcentaje que estas representan en la ejecución de toda la especificación.

D. *Validación de la Especificación*

La Validación es el proceso que incrementa la confianza para lo cual son necesarios casos de prueba que se ejecutarán en el intérprete validando que los resultados obtenidos estén acorde a lo especificado. El primer caso de prueba estuvo referido a validar la correcta creación de examenes. Para ellos se creó el objeto ex con los atributos: `idExamen = 2, idPaciente= 133, fecha (4,10,11),` las señales `bipolar = [2,3,4 ], ampliada=[6,7,5] y precordial= [1,5,8,3,4,6]` y `autorización= 0` (tal como se observa en la línea 2 de la figura. 8. En el segundo caso de prueba se llamó a la función `Fun_Autorizado()` como se muestra en la línea 3 de la figura. 8. Ambos casos de prueba fueron satisfactorios.

```
1:  >> tcov reset
2:  >> create ex:= new Examen(2,133,mk_(4,10,11),[2,3,4],[6,7,5],[1,5,8,3,4,6],0)
3:  >> print ex.Fun_Autorizado()
4:  mk_( "No_autorizado",
5:       [-1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1, -1 ])
6:  >> tcov write vdm.tc
7:  >> rtinfo vdm.tc
8:    100%     1  Examen`Examen
9:      0%     0  Examen`ExamenCompleto
10:   64%     1  Examen`Fun_Autorizado
11:
12:  Total Coverage: 66%
```

Figura 8: Ejecución de Casos de Prueba y Análisis de Cobertura I

Habiendo ejecutado los casos de prueba iniciales, es importante señalar que debemos garantizar que las especificaciones expresen fehacientemente el comportamiento de los requerimientos funcionales, y una forma de lograrlo es mediante la validación de la totalidad de la especificación. Para poder medir que porcentaje de la especificación fue validada con los dos casos de prueba iniciales, se aplicó análisis de cobertura a la especificación formal de la clase `Examen.` En la figura 8, en las líneas 8, 9 y 10 se muestran el porcentaje de cobertura para cada operación. En la línea 7 se muestra el porcentaje total de cobertura en la clase `Examen`. Como se puede apreciar, un 66% de cobertura no es suficiente. Por lo cual se requiere buscar otros casos de prueba para validar las partes de la especificación faltantes.

Si bien el caso en estudio no tiene mucha complejidad (y sería posible validar el total de la especificación sin mayor problema), este nos da una idea resumida de lo que podría ocurrir en especificaciones más complejas y extensas, en donde descubrir que casos de prueba no se han ejecutado no sería una tarea fácil de realizar.

E. *Refinamiento de la Especificación*

Los casos de prueba realizados previamente nos permitieron observar que la operación `ExamenCompleto` no fue validada. Por lo cual se crearon nuevos casos de prueba para validar la operación. De esta manera se está refinando los casos de prueba para cubrir más operaciones en la clase. En los casos de prueba de refinamiento, el primero, estuvo referido a crear el objeto `ex` con `idExamen=2, idPaciente=133, fecha=(4,10,11),` las señales `bipolar=[2,3,4 ], ampliada=[6,7,5] y precordial= [1,5,8,3,4,6]` y `autorización= 1,` como se puede apreciar en la línea 2 de la figura 9. En el segundo caso de prueba se llamo a la función `Fun_Autorizado()` (línea 3 de la Figura 9). Ambos casos de prueba fueron satisfactorios.

Nuevamente se aplicó Análisis de Cobertura a la especificación formal. En las líneas 8,9,10 de la figura 9, se muestran el porcentaje de cobertura para cada operación. Esta vez la operación `ExamenCompleto` fue validada. En la línea 12 se muestra el porcentaje total de cobertura en la `clase Examen: 91%`. Como se puede apreciar, se ha logrado una mejora importante con respecto a la cobertura obtenida en las primeras pruebas.

```
1: >> tcov reset
2: >> create ex:= new Examen(2,133,mk_(4,10,11),[2,3,4],[6,7,5],[1,5,8,3,4,6],1)
3: >> print ex.Fun_Autorizado()
4: mk_( "Autorizado",
5:      [ 2, 3, 4, 6, 7, 5, 1, 5, 8, 3, 4, 6 ])
6: >> tcov write vdm.tc
7: >> rtinfo vdm.tc
8:   100%     1  Examen`Examen
9:   100%     1  Examen`ExamenCompleto
10:   64%     1  Examen`Fun_Autorizado
11:
12: Total Coverage: 91%
```

Figura 9: Ejecución de Casos de Prueba y Análisis de Cobertura II

Tal como se mencionó en la Sección IV, el análisis de Cobertura de la Especificación Formal (Paso 6), y el refinamiento de la especificación mediante la generación de nuevos casos de prueba (Paso 7) se realiza tantas veces sea necesario hasta lograr ejecutar y validar toda la especificación.

F. *Generación de Código Java*

Una vez valida la especificación, el siguiente paso es la generación de código. En este trabajo hemos realizado la generación automática de código Java [7]. El código obtenido se presenta en las Figuras 10 y 11. Para cada clase

VDM++ se genera una clase Java con el mismo nombre (figura 10, línea 10). Para cada miembro de la clase VDM++ se generara el elemento correspondiente en Java con el mismo modificador: entre las líneas 13 a 21 de la figura 10, se ha generado los datos miembros de la clase, los dos primeros con el modificador `public` y el resto con el modificar `private`, y, en las líneas 49, 53 (figura 10) y, líneas 72 y 89 (figura 11) se observa el modificador `public` que acompaña a los constructores y a los métodos respectivamente.

Cuando se genera código Java desde VDM++ es posible generar el cuerpo de los métodos así como código que permita la verificación de las pre-condiciones y post-condiciones. Si la verificación falla se genera un mensaje de error. Todo esto lo podemos ver para el método Examen entre las líneas 53 al 70 (figura 10), para el método `ExamenCompleto` entre las líneas 72 al 87 (figura 11), y el método `Fun_Autorizado` entre las líneas 89 al 108 (figura 11).

El refinamiento de datos del código generado está basado en la Librería VDM Java la cual es implementada en el paquete `jp.co.csk.vdm.toolbox.VDM` (línea 7, figura 10). En esta librería se encuentra una implementación de los tipos de datos VDM++: producto, tupla y registro. Los tipos de datos como `sets`, `sequences`, `maps` son representados por clases que son partes del Java estandard. Además para brindar una implementación de los tipos de datos anteriormente mencionados, la librería de Java VDM provee la clase UTIL [7]. Esta clase contiene métodos auxiliares que son utilizados en la generación de código. Los métodos más importantes son: `clone`, `equals`, `toString`, `RunTime`. El método `clone` clona un valor VDM++. Su aplicación se puede observar en las líneas 62 a 68 (figura 10), 80 y 82 (figura 11).
El método `RunTime` es llamado cuando ocurre un error de ejecución. Su aplicación se observa en las líneas 58 (figura 10) y 75 (figura 11).
La clase Java generada también contiene un comparador estático que implementa la interface `java.util.Comparator` en el JDK. Este es utilizado en una estructura de árbol, e implementa la noción de VDM de igualdad. Su uso se observa en la línea 12 (figura 10). La clase `CGException` maneja los mensajes de error de la librería VDM Java. Cuando se detecta un error. Esto se observa en las líneas 23, 49, 55 (figura 10), 72, y 89 (figura 11).

```
2  // THIS FILE IS AUTOMATICALLY GENERATED!!
3  //
4  // Generated at 2012-05-13 by the VDM++
5  // to JAVA Code Generator
6
7  import jp.co.csk.vdm.toolbox.VDM.*;
8  import java.util.*;
9
10 public class Examen {
11
12   static UTIL.VDMCompare vdmComp = new UTIL.VDMCompare();
13   public Integer idPaciente = null;
14   public Integer idExamen = null;
15   private Tuple fechaExamen = new Tuple(3);
16   public Integer autorizado = null;
17   private Vector bipolar = null;
18   private Vector aumentada = null;
19   private Vector precordial = null;
20   private Vector senalCompleta = null;
21   private Vector senalCompletanoaut = null;
22
23   private void vdm_init_Examen () throws CGException {
24     try {
25
26       autorizado = new Integer(0);
27       bipolar = new Vector();
28       bipolar.add(new Integer(1));
29       bipolar.add(new Integer(1));
30       bipolar.add(new Integer(1));
31       aumentada = new Vector();
32       aumentada.add(new Integer(1));
33       aumentada.add(new Integer(1));
34       aumentada.add(new Integer(1));
35       precordial = new Vector();
36       precordial.add(new Integer(1));
37       precordial.add(new Integer(1));
38       precordial.add(new Integer(1));
39       precordial.add(new Integer(1));
40       precordial.add(new Integer(1));
41       precordial.add(new Integer(1));
42     }
43     catch (Exception e) {
44       e.printStackTrace(System.out);
45       System.out.println(e.getMessage());
46     }
47   }
48
49   public Examen () throws CGException {
50     vdm_init_Examen();
51   }
52
53   public Examen (final Integer Idex, final Integer Idpac, final Tuple fec,
54   final Vector bip, final Vector amb, final Vector prec, final Integer aut)
55   throws CGException {
56
57     if (!this.pre_Examen(Idex, Idpac, fec, bip, amb, prec, aut).booleanValue())
58       UTIL.RunTime("Run-Time Error:Precondition failure in Examen");
59     vdm_init_Examen();
60     {
61
62       fechaExamen = (Tuple) UTIL.clone(fec);
63       idPaciente = UTIL.NumberToInt(UTIL.clone(Idpac));
64       idExamen = UTIL.NumberToInt(UTIL.clone(Idex));
65       bipolar = (Vector) UTIL.ConvertToList(UTIL.clone(bip));
66       aumentada = (Vector) UTIL.ConvertToList(UTIL.clone(amb));
67       precordial = (Vector) UTIL.ConvertToList(UTIL.clone(prec));
68       autorizado = UTIL.NumberToInt(UTIL.clone(aut));
69     }
70   }
```

Figura. 10: Código Java generado a partir de la especificación Parte I

```
public Vector ExamenCompleto () throws CGException {

  if (!this.pre_ExamenCompleto().booleanValue())
    UTIL.RunTime("Run-Time Error:Precondition failure in ExamenCompleto");
  {

    Vector rhs_1 = null;
    Vector var1_2 = null;
    var1_2 = (Vector) bipolar.clone();
    var1_2.addAll(aumentada);
    rhs_1 = (Vector) var1_2.clone();
    rhs_1.addAll(precordial);
    senalCompleta = (Vector) UTIL.ConvertToList(UTIL.clone(rhs_1));
    return senalCompleta;
  }
}

public Tuple Fun_uAutorizado () throws CGException {
  if (autorizado.intValue() == (1)) {

    Vector siautorizado = ExamenCompleto();
    Tuple rexpr_11 = new Tuple(2);
    rexpr_11 = new Tuple(2);
    rexpr_11.SetField(1, new String("Autorizado"));
    rexpr_11.SetField(2, siautorizado);
    return rexpr_11;
  }
  else {

    Vector noautorizado = senalCompletanoaut;
    Tuple rexpr_6 = new Tuple(2);
    rexpr_6 = new Tuple(2);
    rexpr_6.SetField(1, new String("No Autorizado"));
    rexpr_6.SetField(2, noautorizado);
    return rexpr_6;
  }
}
};
```

Figura. 11: Código Java generado a partir de la especificación Parte II

De las pruebas realizadas es importante resaltar que no todas las especificaciones VDM++ pueden ser generadas. Las especificaciones deben cumplir con ciertos requerimientos para que estas puedan ser traducidas en código. Una de las limitaciones es el algoritmo de transformación utilizado.

VI CONCLUSIONES

En este artículo se ha presentado la aplicación de especificaciones formales ligeras en VDM++ como complemento al Modelo Conceptual para poder expresar restricciones adicionales. También se ha presentado la validación de la especificación mediante casos de prueba refinados con análisis de cobertura y su posterior generación de código. Como aporte podemos resaltar que la aplicación de especificaciones formales ligeras permite trabajar un estilo de programación defensiva. Al utilizar análisis de cobertura se puede garantizar que se valide toda la especificación. Además la posibilidad de generar código desde una especificación formal permite incrementar la conformidad entre los requerimientos y el código.

Como caso de aplicación se utilizó un Electrocardiógrafo, en el que en primer lugar se han capturando los requerimientos funcionales en lenguaje natural, para luego convertirlos en una especificación formal ligera VDM++. Los requerimientos fueron definidos de forma clara, sin ambigüedad, y validada de acuerdo a las reglas de VDM++. Para garantizar que toda la especificación está siendo validada, se realizó tantos casos de prueba como fue necesario. Todo esto fue realizado con ayuda del análisis de cobertura, permitiéndonos crear nuevos casos de prueba que validaron las restricciones que no fueron validadas con anterioridad. Finalmente se genero código Java desde la especificación, lo cual permite garantizar la conformidad con los requerimientos de usuario.

REFERENCIAS

[1] B. Beizer, "Black-Box Testing: Techniques for Functional Testing of Software and Systems". John Wiley & Son, Inc. 1995

[2] B. K. Aicherning, "Systematic Black-Box Testin of Computer-Based System through Formal Abstraction Techiniques". Phd Thesis. Technischen Universit¨at Graz, Germany, 2001.

[3] B. Meyer, "Object-oriented Software Construction", Prentice Hall, New York, NY, second edition, 1997.

[4] C. B. Jones, "Formal Methods Light: A Rigorous approach to Formal Methods", IEEE Computer, 29(4): 20-21, April 1996.

[5] CSK SYSTEMS, "VDM++ Toolbox User Manual". Technical Report, 2009.

[6] CSK SYSTEMS, "The VDM++ Language". Technical Report, 2009.

[7] CSK SYSTEMS, "The VDM++ to Java Code Generator". Technica Report 2009

[8] CSK SYSTEMS, "VDM Method Guideline". Technica Report 2009

[9] D. Dubin, "Electrocardiografía Práctica: Lesion Trazado e Interpretacion", 3ra Ed; McGraw Hill Interamericana, 1986

[10] D. Bjørner and C.B. Jones, "Formal Specification and Software Development". Prentice-Hall International, 1982.

[11] D. Martin, T. Kelsey, S. Linton and U.Martin, "Lightweight Formal methods for Computer Algebra System". In IS-SCA '98: International Symposium on Symbolic and Algebraic Computation, University of Rostock, Germany, Auguts, 1998.

[12] D. O. Garcia & L.F. Yavar, "Captacion y Visualizacion de Senales ECG Bipolares: Diseño y Desarrollo". VII Congreso de la Sociedad Cubana de BioIngenieria, 2007.

[13] G. Booch, J. Rumbaugh, and I. Jacobson, "The Unified Modeling Language User Guide", Addison-Wesley, 1998.

[14] I. Sommerville, "Software Engineering", Sixth Edition, Addison Wesley, 2001.

[15] J. Fitzgerald and P.Gorm Larsen, "Modelling Systems Practical Tools and Techniques in Software Development". Cambridge University Press, The Edinburgh Building, Cambridge CB2 2RU, UK, 1998.

[16] J. Fitzgerald, P. Gorm Larsen, P. Mukherjee, N. Plat, and M. Verhoef, "Validated Designs for Object oriented Systems". Springer, New York, 2005.

[17] J. Rumbaugh, I. Jacobson, and G. Booch, "The Unified Modeling Language Reference Manual", Addison-Wesley, 1998.

[18] J. Wartak, *Interpretación de Electrocardiogramas.* 2 Ed., Nueva Editorial Interamericana, 1985

[19] P. Gorm Larsen, J. Fitzgerald, and T. Brookes, "Applying Formal Specification in Industry". *IEEE Software*, 13(3):48–56, May 1996

[20] S. Asaf, E. Marcus and A. Zhiv, "Defining Coverage Views to improve Funtional Coverage Analysis". Proceeding of the 41 annula Conference on Design automation, San Diego, CA, USA, pp: 41-44, 2004

[21] S. Cornett, "Code Coverage Analysis". http://www.bullseye.com/coverage.html

[22] S. M. Easterbrook, R. Lutz, R. Covington, J. Kelly, Y. Ampo, and D. Hamilton. "Experiences using Lightweight Formal Methods for Requirements Modeling". IEEE Transactions on Software Engineering, 24(1), January 1998.

[23] Z. Liu, "Object Oriented Software Development with UML". International Institute for Software Technology, 2002.