Sistema de Control de Acceso a Material de la Biblioteca Universitaria con Smart Cards utilizando Especificaciones Formales

Mayta Rosas Milagros[1], Talavera Díaz Henry[1], Pauca Quispe Diana[1], Villanueva Montoya Luis[1], Vidal Duarte Elizabeth – Mentor[2]
[1]Universidad Nacional de San Agustín de Arequipa, Perú, mmaytar@unsa.edu.pe, htalaverad@unsa.edu.pe, dpaucaq@unsa.edu.pe, lvillanuevamo@unsa.edu.pe
[2]Universidad Nacional de San Agustín de Arequipa, Perú, evidald@unsa.edu.pe

***Abstract*– La biblioteca de la Universidad Nacional de San Agustín de Arequipa actualmente no cuenta con un sistema automatizado para proporcionar el servicio adecuado a los estudiantes de manera que no se cubren las necesidades de eficiencia en el uso de este servicio. En este trabajo se propone la implementación de un Sistema de Control de Acceso al material de la biblioteca universitaria con el uso de Smart Cards, identificando los principales requerimientos del sistema, además se muestra el uso de especificaciones formales que reducen la ambigüedad de los requerimientos y muestran la cobertura total validando el modelo propuesto.**

Keywords—Especificaciones Formales, VDM++, Smart Cards, Validación

I. Introducción

En lo últimos años se ha demostrado que el uso de especificaciones formales en sistemas reales no aumenta el costo de desarrollo del software y asegura una mejora en la calidad del producto. Estas son aplicadas en su mayoría a sistemas de transporte [1], [2], pero también se utilizan en sistemas de información y telecomunicaciones, control de plantas, acceso y seguridad [10]; Con el fin de reducir la ambigüedad en los requerimientos.

En este trabajo se presenta el uso de especificaciones formales en los requerimientos de un Sistema de Control de Acceso al material de una biblioteca universitaria con Smart Cards, principalmente para incrementar la certidumbre de estos, de la misma manera se presenta la validación de la especificación formal con una cobertura total del modelo propuesto.

El resto de artículo está organizado de la siguiente manera. En la Sección II se presentan algunos de los trabajos relacionados. La sección III muestra una definición de Smart Card características generales, ventajas y desventajas. En la sección IV se presenta una definición de especificaciones formales, la definición del lenguaje VDM++, clases, tipos, invariantes y operaciones, también se presenta la herramienta utilizada VDM++Toolbox. En la sección V se presenta la propuesta con el planteamiento del problema, la identificación de los principales requerimientos, el diagrama de clases y la especificación formal del modelo. La sección VI muestra la validación total del modelo mediante la clase Test y los resultados obtenidos. Finalmente, en la sección VII se presentan las conclusiones finales del trabajo realizado.

II. Trabajos Relacionados

La especificación formal ha sido una técnica valiosa durante el desarrollo de software y se han aplicado en muchos tipos de sistemas, por ser parte importante en el éxito o fracaso de su desarrollo. Como se muestra en la investigación de Jo, Yoon y Hwag [1], que realizaron la incorporación del lenguaje formal Z para un Sistema de Control de Trenes con el fin de proporcionar mayor seguridad a la aplicación. El trabajo de Zhao y Rozier [2], propone el uso de técnicas de verificación formal para asegurar que no existan fallas con consecuencias fatales en la aplicación de control de tráfico aéreo que ellos evalúan.

Katayama, et al. [3], utiliza VDM++ en las especificaciones formales y a partir de estas se desarrolla una herramienta de generación de tablas de decisión. El trabajo de Wolff [4] presenta un caso de especificación de requerimientos de un sistema de autodefensa utilizado a bordo de aviones de combate.

Pero el uso de especificaciones formales ya no se aplica sólo en sistemas considerados como críticos sino se ha expandido a otro tipo de aplicaciones. Actualmente diversas instituciones están apostando por el uso de Smart Cards, a consecuencia de esto surge la necesidad de que los usuarios confíen en que los detalles y datos de su vida privada no se transmiten a terceros. Diversas investigaciones como la de Catano y Huisman [5] desarrollan una aplicación de monedero electrónico, utilizando una Smart Card donde la especificación formal permitió la detección de diversos errores y posibilidades de mejora en el código fuente. Liu, McDermid y Chen [6] presentan un caso de estudio de detección de errores de un cajero automático que funciona con una Smart Card utilizando de forma rigurosa y sistemática las especificaciones formales. Estos autores han concluido que la mejor manera de garantizar la seguridad y la fiabilidad de los sistemas es mediante el uso de técnicas formales para la especificación y posterior verificación de las aplicaciones de Smart Card.

La automatización de diversos procesos en muchas instituciones, plantea mayores requisitos en los aspectos administrativos de estas, como son la identificación de usuarios, permisos y accesos a diversos materiales. El trabajo de Lv [7], desarrolla una planificación de un sistema de Smart Cards para el control de acceso a los campus de las universidades. Subhash, Gayathri y Gayathri [8], propone el uso de Smart Cards con tecnología RFID para el acceso a diferentes aplicaciones entre ellas los campus universitarios, demuestra que las Smart Cards son una alternativa para viabilizar esta situación.

Nuestro trabajo propone la especificación formal de requisitos, utilizando el lenguaje VDM++, de la aplicación que hará uso de Smart Cards en el campus de la Universidad Nacional de San Agustín para el control de acceso a material bibliográfico, de tal forma que garantizamos tanto los detalles de seguridad de los usuarios, así como su correcto funcionamiento, además de aprovechar las facilidades que VDM++ nos ofrece en métodos formales.

III. SMART CARD

A. *Definición de Smart Card*

No existe una definición formal para una Smart Card. La descripción que le da la industria es una "tarjeta" de plástico o dispositivo de dimensiones normalizadas, al igual que una tarjeta de crédito, que contiene un chip integrado [12], [18], [20]. La Smart Card se conecta a un lector con el contacto físico directo o usando una interfaz sin contacto de radiofrecuencia [20].

El chip puede comprender un sistema operativo con software, datos, programas pequeños y se puede proteger contra el acceso externo o la alteración mediante un sistema de seguridad criptográfica [12]. Además, tiene la capacidad de almacenar gran cantidad de datos, llevar a cabo sus propias funciones, como el cifrado y la autenticación o interactuar de forma inteligente con un lector de tarjetas. La tecnología que utiliza se ajusta a las normas internacionales ISO 7816 e ISO/IEC 14443 [18], [20].

B. *Características*

Una de las características más relevantes de una Smart Card es la seguridad para la información, esta característica la hace una de las mejores alternativas para llevar a cabo no solo transacciones financieras sino otras muchas aplicaciones, tales como sistemas de acceso y control, aplicaciones en el sector bancario, sistemas de transporte masivo, industria de las telecomunicaciones, aplicaciones en el sector salud, aplicaciones en el ámbito comercial para diferentes tipos de industria y para la autenticación de usuarios mediante firmas digitales con el fin de controlar el acceso a diferentes páginas web y realizar transacciones online de forma segura [19], [20], [21].

Según M'Chirgui [12], y García [19], desde el punto de vista tecnológico existen dos tipos de Smart Card teniendo en cuenta la función que realizan: Una es la tarjeta de memoria que contienen un chip simple con carencia de circuito de procesamiento. Estas tarjetas se usan principalmente para aplicaciones y sistemas de pago de servicios o telecomunicaciones y la mayoría pueden ser programadas usando cualquier lector de Smart Card [19].

El otro tipo de tarjetas son las que poseen un microprocesador como el centro operativo de la tarjeta, la mayoría de fabricantes utilizaron la arquitectura convencional de von Neumann para este tipo de tarjetas [19]. El tamaño y la potencia del chip determinan su capacidad de almacenamiento y procesamiento. Las tarjetas de microprocesador son bastante flexibles en sus aplicaciones. Estos tipos de tarjetas son útiles cuando se necesita almacenar una clave digital o una identidad de manera segura, así como para aplicaciones que necesitan realizar funciones criptográficas especializadas [12], [19].

Desde la perspectiva de las características físicas, hay dos categorías generales [12], [20]: las tarjetas con contacto donde estas deben ser insertadas en un lector de tarjetas. Esta tarjeta contiene puntos de contacto físicos en su superficie que permiten la transmisión de datos e información de estado entre la tarjeta y el lector, y las tarjetas sin contacto que solo necesitan estar cerca de un lector. Tanto la tarjeta y el lector tienen antenas de radio de alta frecuencia con las cuales realizan el enlace sin contacto.

C. *Ventajas y Desventajas*

Ventajas: Las Smart Cards son de mercado innovador y creciente, en la actualidad tienen más y diversas aplicaciones. La implementación del uso de Smart Cards en algunos sistemas aumenta su eficiencia reduciendo y facilitando el tiempo de las operaciones, además proporcionan seguridad de almacenamiento de datos y las más sofisticadas pueden almacenar software complejo. Además, se adhiere a estándares internacionales lo que garantiza múltiples proveedores y comparativa de precios, tiene un vasto historial en aplicaciones exitosas en el mundo real y sobre todo presenta larga vida útil de hasta 10.000 lecturas/escrituras antes de la falla [12], [21].

Desventajas: Según las capacidades y características de las Smart Cards estas pueden tener altos costos, además se debe añadir el costo del hardware de reconocimiento, su implementación y mantenimiento. También se debe resolver diferentes y tediosas cuestiones jurídicas y de política relacionados con la confidencial y la protección de los consumidores [12].

IV. ESPECIFICACIONES FORMALES

A. *Definición*

La especificación formal es la expresión que se basa en un lenguaje con sintaxis y semántica matemáticamente definidas, usando conceptos como abstracción y composición. Esta

expresión representa una serie características que el sistema de cumplir y satisfacer, y se utilizan para especificar y verificar tanto el hardware como los sistemas de software [10], [9]. Estas características abarcan desde los objetivos de alto nivel, requisitos no funcionales entre otros componentes [10].

El objetivo del uso de especificaciones formales es reducir al mínimo la ambigüedad y la incertidumbre propias de cualquier especificación hecha en lenguaje natural, utilizando la precisión de las notaciones matemáticas y las expresiones lógicas [9].

B. VDM++

Es un lenguaje de especificación formal orientado a objetos que fue desarrollado a partir de VDM-SL, lenguaje de especificación basado en modelos. En VDM++ el sistema se representa como un conjunto de clases que poseen constantes, variables, operaciones y funciones. VDM++ permite manejar concurrencia y sistemas distribuidos en tiempo real. La semántica y la sintaxis de VDM++ es fácilmente comprensible y cercana a un lenguaje de programación de alto nivel por lo que es muy fácil adoptar. VDM ++ es un enfoque orientado a objetos que proporciona soporte y sincronización de simultaneidad [11]. Presentamos conceptos y sintaxis de VDM++ utilizados en nuestro trabajo y tal como se explica en [14], [15], [16].

1) Definición de clases: Los modelos en VDM++ están conformados por un conjunto de clases. Para definir una clase se utiliza la palabra reservada *class* seguida del nombre de esta. Una clase en VDM++ tiene las partes que se muestra en la Fig. 1[15]. Como es notorio las variables de instancia se encuentra al inicio en el primer bloque, luego están los Tipos, valores y Operaciones todo ellos agrupados en el segundo bloque de definiciones, VDM++ permite el uso de hilos y sincronización que se encuentran en el tercer y cuarto bloque de la clase respectivamente.

2) Tipos: VDM++ proporciona tipos básicos y tipos compuestos para poder ser utilizados. En los tipos básicos se encuentran los tipos booleanos, naturales, reales y char; mientras que en los tipos compuestos están los conjuntos, secuencias, entre otros. Cada uno de estos tipos cuenta con operaciones predefinidas.

En este trabajo se utilizan los tipos básicos natural (nat), enteros (int), booleanos (bool) y carácter (char). El tipo Secuencias (seq of) es el tipo compuesto utilizado.

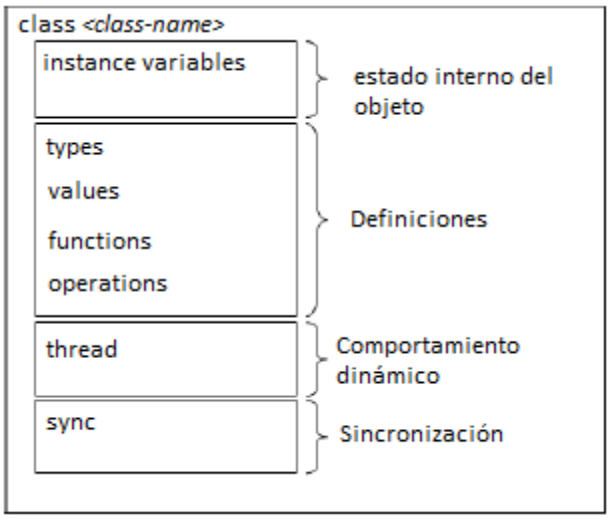

Fig. 1 Estructura de una clase en VDM++.

3) Invariantes: Una invariante es la restricción de valores no permitidos en las variables de instancia de una clase. Es una condición que debe contener siempre un estado de objeto [14].

En VDM++ se utiliza la palabra reservada *inv* luego de la definición de las variables de instancia. Como se presenta en la Fig. 2.

```
class <nombre-clase>
  instance variables
  definición 1;
  definición 2;
  ...
  definición n;
  inv expresión utilizando las variables de instancia

end <nombre-clase>
```

Fig. 2 Especificación de una invariante.

La implementación de una invariante es una función de VDM++ que recibe el tipo de parámetro que quiere validarse y devuelve siempre un *bool*, en la Fig. 3, se muestra primero la implementación de una función general y luego una función invariante.

```
class <nombre-clase>
  functions
  nombreFunción: tipo parámetros +> tipo retorno
  nombreFunción (parámetros)==
      sentencias
  nombreInvariante: tipo parámetros +> bool
  nombreInvariante (parámetros)
      sentencias
end <nombre-clase>
```

Fig. 3 Especificación de una función y una función invariante.

4) Operaciones: Las operaciones son las que definen los algoritmos en VDM++. Estas pueden ser implícitas como por ejemplo las pre-condiciones y post-condiciones o explicitas, pero en la herramienta deben ser definidas de forma explícita todas las operaciones, incluso las pre y post condiciones [14].

En la Fig. 4, tomada de [14], se puede observar la sintaxis de algunas de las operaciones en VDM++.

```
class <nombre-clase>
  operations
  nombreOperación: tipo parámetros ==> tipo retorno
  nombreOperación (parámetros)==
          sentencias
  pre expresión
  post expresión

end <nombre-clase>
```

Fig. 4 Especificación de operaciones.

C. *La Herramienta: VDM++ Toolbox*

VDM++ Toolbox [17], es una herramienta integrada que soporta el análisis de modelos VDM++ vía análisis sintáctico, análisis de tipos, generación de condiciones de integridad y pruebas. Además, permite la especificación de pre-condiciones, post-condiciones e invariantes.

La herramienta posee características que permiten generar especificación VDM++ desde diagramas UML y viceversa. Adicionalmente tiene un intérprete que permite validar las especificaciones. La Fig. 5, muestra la interfaz de trabajo de la herramienta. La Fig. 6, muestra el intérprete que fue utilizado para la validación mediante la generación de casos de prueba.

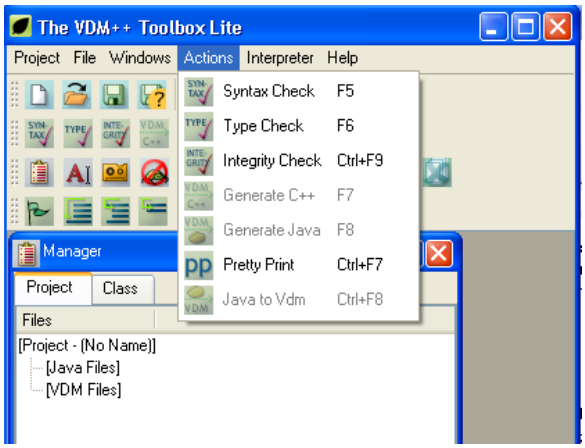

Fig. 5 Interface de VDM++ Toolbox.

Las especificaciones son realizadas con un propósito: generalmente para lograr una mejor comprensión del comportamiento deseado de un sistema propuesto, 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.

VDM++ Toolbox provee soporte para validación a través de animación y pruebas utilizando un intérprete. El intérprete permite ejecutar partes de la especificación en valores seleccionados. En la Fig. 6, se muestra el intérprete de la herramienta.

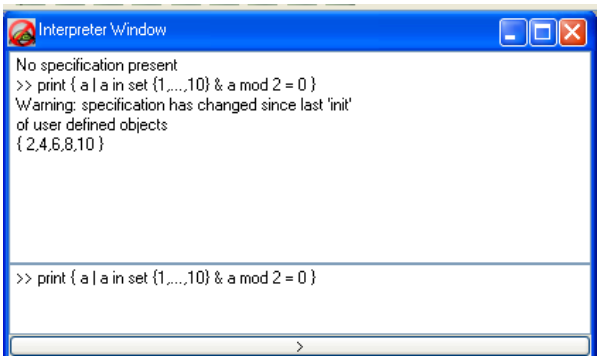

Fig. 6 Interprete de VDM++ Toolbox.

V. Propuesta

A. *Planteamiento del problema*

La Universidad Nacional de San Agustín, Arequipa – Perú, que a la fecha tiene 47 escuelas profesionales, y alrededor de 27000 estudiantes, cuenta con tres bibliotecas generales ubicadas en el campus universitario [13].

En la actualidad para acceder a este servicio los estudiantes deben tramitar un carnet de biblioteca que contiene impreso: Nombres completos, código y una fotografía del estudiante como se muestra en las Fig. 7 y Fig. 8, Sin embargo, para acceder a los libros debe dejar su carnet en la biblioteca y devolver el mismo como máximo en dos días no quedando registro alguno de los libros que son prestados y devueltos.

Esta situación genera una problemática ya que se pone en riesgo principalmente la pérdida de los libros y los carnets bibliotecarios.

Nuestra propuesta es la automatización del servicio de biblioteca para los estudiantes, mediante la implementación de un sistema de Smart Cards que permitirá a los administradores llevar un mejor control y registro fechado de todos los libros prestados y devueltos por cada estudiante, de esta manera mantener íntegro y accesible este servicio fundamental en la universidad.

Fig. 7 Carnet de biblioteca actual (Anverso).

Fig. 8 Carnet de biblioteca actual (Reverso).

B. Principales Requerimientos

De la problemática descrita se han identificado los siguientes requerimientos:

R1 Se tendrá un registro de los materiales que son prestados y devueltos.

R2 Solo se permitirá 3 libros prestados simultáneamente a cada Smart Card.

R3 La Smart Card no permitirá que se retire ningún libro más, si éste último excede la cantidad de libros máximos prestados por Smart Card.

R4 Se contará con un registro de todas las Smart Card.

R5 Realizar préstamo de libro.

R6 Realizar devolución de libro.

R7 Cada código de Smart Card debe ser único.

C. Diagrama de Clases y Especificación formal en VDM++

El diagrama de clases propuesto para el sistema de acceso se muestra en la Fig. 9.

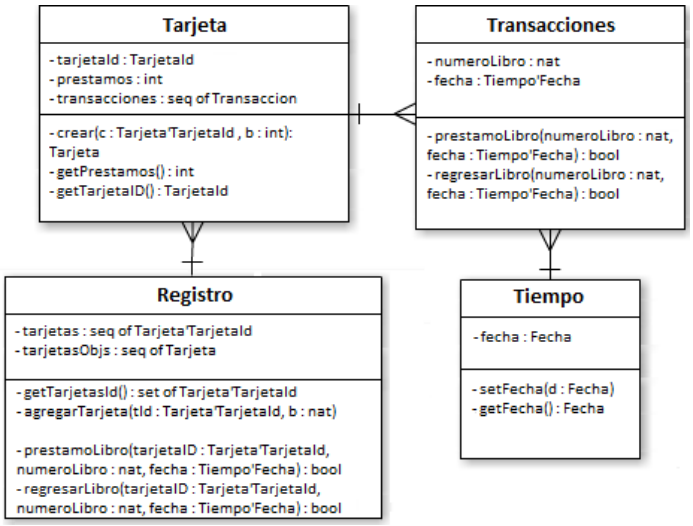

Fig. 9 Diagrama de Clases.

En las Fig. 10, Fig. 11, Fig. 12 y Fig. 13 se presenta la especificación formal en VDM++ correspondiente a los requerimientos propuestos.

```
1: class Card
2:
3: values limite : nat = 3;
4:
5: types
6:  public CardId = nat;
7:  Transaccion :: fecha : Reloj`Fecha
8:          cardId : Card`CardId
9:          numeroLibro : nat;
10:
11: instance variables
12:  cardId : CardId;
13:  prestamos : int := 0;
14:  inv prestamosInvariante(prestamos);
15:  transacciones : seq of Transaccion := [];
16:
17:
18: operations
19:  public
20:  Crear : Card`CardId ==> Card
21:  Crear(c) ==
22:    (cardId := c;
23:    prestamos := 0;
24:    return self);
25:
26:  public
27:  GetPrestamos : () ==> int
28:  GetPrestamos() ==
29:    return prestamos;
30:
31:
32:  public
33:  GetCardId : () ==> CardId
34:  GetCardId() ==
35:    return cardId;
36:
```

Fig. 10 Implementación de la clase Card en VDM++ (parte 1).

```
37:  public prestarLibro : nat * Reloj`Fecha ==> bool
38:  prestarLibro(numerolibro,fecha) ==
39:    let transaccion = mk_Transaccion(fecha,cardId,numerolibro)
40:    in
41:    if prestamos < limite
42:    then
43:        (prestamos := prestamos + 1;
44:        transacciones := transacciones^[transaccion];
45:        return true;)
46:    else
47:        return false;
48:
49:
50:  public devolverLibro : nat * Reloj`Fecha ==> bool
51:  devolverLibro(numerolibro,fecha) ==
52:    let transaccion = mk_Transaccion(fecha,cardId,numerolibro)
53:    in
54:    if prestamos > 0
55:    then
56:        (prestamos := prestamos - 1;
57:        transacciones := transacciones^[transaccion];
58:        return true;)
59:    else
60:        return false;
61:
62:  functions
63:  prestamosInvariante: int +> bool
64:  prestamosInvariante(ln) ==
65:    ln <= limite and ln >= 0
66:
67:  end Card
```

Fig. 11 Implementación de la clase Card en VDM++ (parte 2).

```
1:  class Registro
2:
3:  instance variables
4:      cards : seq of Card`CardId := [];
5:      cardsObjs : seq of Card := [];
6:
7:  operations
8:
9:  public GetCardIds: () ==> seq of Card`CardId
10:  GetCardIds() == return cards;
11:
12:
13:  public GetCardPrestamos: () ==> seq of int
14:  GetCardPrestamos() ==
15:    return [cardsObjs(i).GetPrestamos() | i in set inds cardsObjs];
16:
17:  public AgregarCard : Card`CardId ==> ()
18:  AgregarCard(cId) ==
19:    (cards := cards ^ [cId] ;
20:    let newCard = (new Card()).Crear(cId)
21:    in
22:
23:    cardsObjs := cardsObjs ^ [newCard];)
24:  pre cId not in set elems cards;
25:
26:  public prestarLibro : Card`CardId * nat * Reloj`Fecha ==> bool
27:  prestarLibro(cardId,numerolibro,fecha) ==
28:    (let CardSeleccionada = hd([cardsObjs(i) | i in set inds cardsObjs & cardsObjs(i).GetCardId() = cardId])
29:    in
30:    return CardSeleccionada.prestarLibro(numerolibro,fecha);)
31:  pre cardId in set elems cards;
32:
33:  public devolverLibro : Card`CardId * nat * Reloj`Fecha ==> bool
34:  devolverLibro(cardId,numerolibro,fecha) ==
35:
36:    (let CardSeleccionada = hd([cardsObjs(i) | i in set inds cardsObjs & cardsObjs(i).GetCardId() = cardId])
37:    in
38:    return CardSeleccionada.devolverLibro(numerolibro,fecha);)
39:  pre cardId in set elems cards;
40:
41:
42:  end Registro
```

Fig. 12 Implementación de la clase Registro en VDM++.

```
1:  class Reloj
2:
3:  types
4:      public Fecha = seq of char;
5:
6:  instance variables
7:      fecha : Fecha := [];
8:
9:  operations
10:     public
11:     SetFecha : Fecha ==> ()
12:     SetFecha(d) ==
13:         fecha := d;
14:
15:     public
16:     GetFecha : () ==> Fecha
17:     GetFecha() ==
18:         return fecha;
19:  end Reloj
```

Fig. 13 Implementación de la clase Reloj en VDM++.

Notese que la clase Transacciones, definida en el diagrama como una clase concreta, está implementada dentro de la clase Card como un tipo (type).

A continuación se presenta una descripción de la especificación formal para cada requerimiento identificado.

R1 está definido en la clase "Card", (Fig. 10), en la línea 15, como una secuencia de "Transacciones", que a su vez está definida en la misma clase en las líneas 8 a la 10.

R2 está definido en la clase "Card" (Fig. 10 y Fig. 11), en la línea 4, como un valor, y en la variable de instancia "prestamos"; en la línea 14 se respalda lo anterior con la invariante "prestamosInvariante" definida en las líneas 63 a la 65.

R3 está definido en la clase "Card" (Fig. 11), en las líneas 41 y 54, en comparaciones de la variable "prestamos" con el valor definido "limite".

R4 está definido en la Clase "Registro" (Fig. 12) como una secuencia de "Card", en la línea 5.

R5 y **R6** están definidos en la clase "Card" (Fig. 11) en las líneas 37 a la 47 y 50 a la 60 respectivamente, en las operaciones "prestarLibro" y "devolverLibro".

R7 está definido en la precondición de la operación "AgregarCard" en la clase "Registro" (Fig. 12), en la línea 24.

VI. Validación

Para llevar a cabo la validación total del modelo está implementada la clase Test mostrada en la Fig. 14, En las líneas 22 y 23 se observa la creación de objetos Registro y Reloj, en la línea 43 se establece la fecha y en las líneas 31 a la 35, mediante el objeto registro se añaden nuevas Cards con su respectivo id.

```
18: public
19: test0: () ==> ()
20: test0() ==
21:
22:   (r1 := new Registro();
23:    d1 := new Reloj(););
24:
25: public
26: test1: () ==> seq of Card`CardId
27: test1() ==
28:
29:   (test0();
30:   test2();
31:   r1.AgregarCard(1);
32:   r1.AgregarCard(2);
33:   r1.AgregarCard(3);
34:   r1.AgregarCard(4);
35:   r1.AgregarCard(5);
36:   test3();
37:   test4();
38:   return r1.GetCardIds());
39:
40: public
41: test2 : () ==> ()
42: test2() ==
43:   d1.SetFecha("29/09/2016");
```

Fig. 14 Fragmento de la clase Test.

En la misma clase posteriormente se hacen múltiples llamadas a los métodos prestarLibro() y devolverLibro() de la clase Registro, entre las líneas 49 a la 83. También se hacen actualizaciones de las fechas en las líneas 67 y 84 que se observan en la Fig. 15.

```
49:   r1.prestarLibro(1,2,d1.GetFecha()),
50:   r1.prestarLibro(1,2,d1.GetFecha()),
51:   r1.prestarLibro(1,12,d1.GetFecha()),
52:   r1.prestarLibro(1,8,d1.GetFecha()),
53:   r1.prestarLibro(1,2,d1.GetFecha()),
54:   r1.prestarLibro(1,20,d1.GetFecha()),
55:   r1.prestarLibro(3,2,d1.GetFecha()),
56:   r1.prestarLibro(3,2,d1.GetFecha()),
57:   r1.prestarLibro(3,2,d1.GetFecha()),
58:   r1.prestarLibro(3,2,d1.GetFecha()),
59:   r1.prestarLibro(3,2,d1.GetFecha()),
60:   r1.prestarLibro(3,2,d1.GetFecha()),
61:   r1.prestarLibro(3,1,d1.GetFecha()),
62:   r1.prestarLibro(3,1,d1.GetFecha()),
63:   r1.prestarLibro(3,1,d1.GetFecha()),
64:   r1.prestarLibro(3,1,d1.GetFecha()),
65:   r1.prestarLibro(3,1,d1.GetFecha()),
66:   r1.prestarLibro(3,1,d1.GetFecha())}
67:   in d1.SetFecha("30/09/2016");
68:
69: public
70: test4 : () ==> ()
71: test4() ==
72:   (let verif = {r1.devolverLibro(3,20,d1.GetFecha()),
73:   r1.prestarLibro(3,2,d1.GetFecha()),
74:   r1.prestarLibro(3,2,d1.GetFecha()),
75:   r1.prestarLibro(3,2,d1.GetFecha()),
76:   r1.prestarLibro(3,1,d1.GetFecha()),
77:   r1.prestarLibro(3,1,d1.GetFecha()),
78:   r1.devolverLibro(3,50,d1.GetFecha()),
79:   r1.devolverLibro(3,50,d1.GetFecha()),
80:   r1.devolverLibro(3,50,d1.GetFecha()),
81:   r1.devolverLibro(3,50,d1.GetFecha()),
82:   r1.devolverLibro(3,50,d1.GetFecha()),
83:   r1.devolverLibro(3,50,d1.GetFecha())}
84:   in d1.SetFecha("01/10/2016");
```

Fig. 15 Fragmento de la clase Test.

La clase Test fue ejecutada en el intérprete de la herramienta. La salida de la ejecución se muestra en la Fig. 16, en donde las 11 primeras líneas corresponden a las llamadas del script ejecutado y sus salidas; entre las líneas 12 a la 31 se muestra la cobertura de los métodos en porcentaje y el conteo de llamadas a cada método.

El porcentaje al 100% significa que el método mencionado en la línea tuvo una cobertura total, por ejemplo, en la línea 30 de la Fig. 16, se muestra una cobertura del 100% en el método devolverLibro de la clase Registro, esto quiere decir que todas las líneas del método fueron ejecutadas en esta llamada.

```
1  Initializing specification ...
2
3  done
4  tcov reset
5  create t := new Test()
6  print t.run()
7  [ 1, 2, 3, 4, 5 ]
8  print t.test5()
9  [ 3, 0, 0, 0, 0 ]
10 tcov write vdm.tc
11 rtinfo vdm.tc
12   100%     5 Card`Crear
13   100%   155 Card`GetCardId
14   100%     5 Card`GetPrestamos
15   100%    24 Card`prestarLibro
16   100%     7 Card`devolverLibro
17   100%    37 Card`prestamosInvariante
18   100%     1 Test`run
19   100%     1 Test`test0
20   100%     1 Test`test1
21   100%     1 Test`test2
22   100%     1 Test`test3
23   100%     1 Test`test4
24   100%     1 Test`test5
25   100%    31 Reloj`GetFecha
26   100%     3 Reloj`SetFecha
27   100%     1 Registro`GetCardIds
28   100%     5 Registro`AgregarCard
29   100%    24 Registro`prestarLibro
30   100%     7 Registro`devolverLibro
31   100%     1 Registro`GetCardPrestamos
32
33 Total Coverage: 100%
```

Fig. 16 Porcentaje de cobertura del modelo mediante la ejecución de la clase Test.

En las líneas 29 y 30 de la Fig. 16, se ven las llamadas a prestarLibro y devolverLibro, hechas en la clase test, Fig. 15, entre las líneas 49 a 84.

Estas llamadas sumadas a las creaciones de las Cards, en las líneas 31 a 35 de la Fig. 12, y con salida de cobertura en las líneas 14 y 28 de la Fig. 16, muestran un total de cobertura a la llamada de la invariante en la línea 17 de la Fig. 16.

En general la cobertura del modelo mediante las pruebas de la clase Test es total.

VII. Conclusiones

En este trabajo se ha presentado como principal aporte la validación de la especificación formal obtenida a partir de los requerimientos identificados en la propuesta planteada, un "Sistema de Control de Acceso a Material de Biblioteca Universitaria con Smart Cards" ante un problema que surge dentro de nuestra comunidad universitaria, con el uso de especificaciones formales en VDM++ y mediante la aplicación de una clase Test con casos de prueba ejecutados en el intérprete de la herramienta VDM++Toolbox, cuyos resultados demostraron la validez acorde a lo especificado. De esta manera se ha incrementado la confiabilidad del modelo propuesto y se ha reducido la ambigüedad que comúnmente genera el lenguaje natural en la identificación de los requerimientos. Por lo tanto, la futura implementación del sistema de control de acceso al material de biblioteca con Smart Cards, se reconoce como una solución viable ante la problemática anteriormente expuesta.

Reconocimiento

En reconocimiento a Elizabeth Vidal, nuestra mentora por su constante apoyo y a la Universidad Nacional de San Agustín.

Referencias

[1] H. Jo, Y. Yoon, and J. Hwang, "Analysis of the Formal Specification Application for Train Control Systems," Journal of Electrical Engineering & Technology, vol. 4, no.1, pp. 87-92, 2009.

[2] Y. Zhao and K. Y. Rozier, "Formal specification and verification of a coordination protocol for an automated air traffic control system," Science of Computer Programming, vol. 96, pp. 337-353, 2014.

[3] T. Katayama, K. Nishikawa, Y. Kita, H. Yamaba, K. Aburada and N. Okazaki, "Prototype of a Decision Table Generation Tool from the Formal Specification," Journal of Robotics, Networking and Artificial Life, vol. 2, no. 3, pp. 25-208, December 2015.

[4] S. Wolff, "Using Executable VDM++ Models in an Industrial Application-Self-defense System for Fighter Aircraft," Technical Report Electronics and Computer Engineering, vol. 1, no. 1, 2015.

[5] N. Catano and M. Huisman, "Formal specification and static checking of Gemplus' electronic purse using ESC/Java," in International Symposium of Formal Methods Europe, Springer Berlin Heidelberg, pp. 272-289, July 2002.

[6] S. Liu, J. A. McDermid and Y. Chen, "A rigorous method for inspection of model-based formal specifications," IEEE Transactions on Reliability, vol. 59, no.4, pp. 667-684, 2010.

[7] W. C. Lv, "Design of Campus Smart Card System. In Applied Mechanics and Materials," in Applied Mechanics and Materials, Trans Tech Publications, vol. 347, pp. 3915-3918, 2013.

[8] J. Subhash, S. Gayathri and D. Gayathri, "Multipurpose Card Using Rfid Technology," International Journal of Engineering and Computer Science, vol. 5, pp. 16026-16028, 2016.

[9] S. J. Jang, J. Ryoo and C. Lee, "Design of Software Security Verification with Formal Method Tools," IJCSNS, vol. 6, no. 9B, pp. 163-167, September 2006.

[10] E. Serna and A. Serna, "La especificación formal en contexto: actual y futuro," Revista chilena de ingeniería, vol. 22, no. 2, pp. 243-256, 2014.

[11] T. Pandey and S. Saurabh, "Comparative analysis of formal specification languages Z, VDM, B," International Journal of Current Engineering and Technology E-ISSN, vol. 5, no. 3, pp. 2277-4106, June 2015.

[12] Z. M'Chirgui, "The economics of the smart card industry: towards coopetitive strategies," Economics of Innovation and New Technology, vol. 14, no. 6, pp. 455-477, 2005.

[13] Universidad Nacional de San Agustín, "Reseña Histórica", [Online]. Available: http://www.unsa.edu.pe/.

[14] E. Vidal-Duarte, "Aplicación y Validación de Especificaciones Formales Ligeras en el Modelo Conceptual: Reduciendo la ambigüedad e incrementando la Conformidad entre los Requerimientos y el Código," 2012.

[15] A. Müller, "VDM—The Vienna Development Method," Bachelor thesis in "Formal Methods in Software Engineering", Research Institute for Symbolic Computation (RISC), Johannes Kepler University Linz, Austria, 2009.

[16] E. Durr and J. Van Katwijk, "VDM++, A Formal Specification Language for Object-Oriented Designs", In CompEuro'92.'Computer

Systems and Software Engineering', Proceedings. IEEE, pp. 214-219, 1992.
[17] CSK SYSTEMS, "VDM++ Toolbox User Manual". Technical Report, 2009.
[18] E. Vargas-García, V. Trujillo-Olaya, J. Velasco-Medina, "Diseño de un carné de identificación universitario mediante tarjetas inteligentes," Revista Ingeniería y Competitividad, vol. 9, no. 2, pp. 93-104, 2007.
[19] C. Henriquez, "Sistema de control de acceso basado en Java Cards y Hardware libre,", Revista Prospectiva de la Universidad del Caribe, vol. 8, no. 2, pp. 63-68, 2010.
[20] Smart Card Alliance, [Online]. Available: http://www.smartcardalliance.org/.
[21] M. T. Ortega, "Implementación de Tarjeta Inteligente Java Card para el Control de Acceso a Instalaciones."