Pruebas basadas en máquinas de estado finitas (FSM)

HENRY ROBERTO UMAÑA A., Profesor Asociado, hrumana@unal.edu.co
MIGUEL ANGEL CUBIDES G., Investigador, macubidesgo@unal.edu.co

*ColSWE, **Col**ectivo de investigación en ingeniería de **S**oft**W**ar**E**, Ingeniería de Sistemas e Industrial. Universidad Nacional de Colombia.*

Resumen

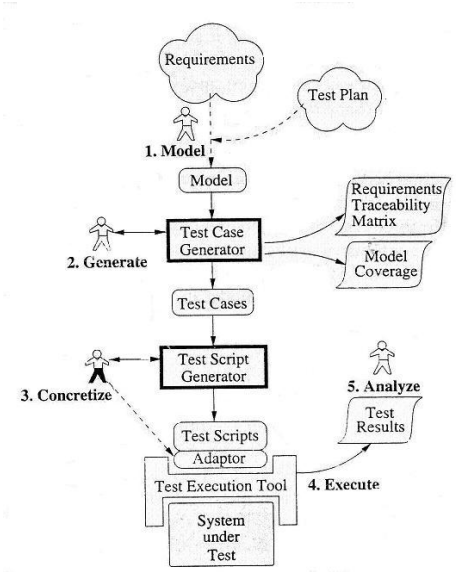
Este capítulo presenta la implementación de pruebas basadas en modelos de máquinas de estado finitas (Finite State Machines FSM). Para el efecto se utiliza la librería ModelJUnit del mismo autor de JUnit, Mark Utting, que explora las pruebas basadas en FSM. ModelJUNit permite que los modelos FSM sean escritos en Java y luego generar y ejecutar pruebas desde estos modelos. Como caso de estudio se toma una aplicación básica de cajero automático (ATM) con pocas restricciones y se modela el sistema como una FSM y se aplica luego ModelJUnit, generándose métricas de cobertura y una representación gráfica de la FSM.

Abstract

This chapter introduces the implementation of Model-Based Testing using Finite State Machines (FSM) as a model of the system's behavior. We are using ModelJUnit library, offered by Mark Utting, the same author of JUnit. ModelJUnit allows FSM models to be written in Java and then build and run tests from these models. As a case study we utilized a basic application of an Automatic Teller Machine (ATM) with few restrictions. Then the system is modeled as a FSM and ModelJUnit is applied, generating coverage metrics and graphical representation of the FSM

1. Introducción

El proceso de pruebas basadas en modelos puede ser dividido en los siguientes 5 pasos [1]:



Gráfica 1. Model Based Testing. Tomado de Utting [1]

Modelar el SUT (System Under Test): Se trata de construir un modelo del sistema que deseamos probar. Usualmente este modelo abarca solamente los detalles que deseamos probar, no necesariamente toda la funcionalidad o estructura del sistema.

Generar casos de pruebas abstractos desde el modelo: Aquí se decide el criterio de selección para la generación de los casos de prueba, por ejemplo en términos de cobertura. Se especifica que son casos de prueba abstractos en la medida en que la salida es una secuencia de operaciones desde el modelo.

Concretizar los casos de prueba abstractos: Consiste en transformar los casos de prueba abstractos en casos de prueba concretos, es decir en un script o código de prueba ejecutable en un ambiente de ejecución determinado.

Ejecutar los casos de prueba concretos: Con pruebas basadas en modelo en línea (online), las pruebas se ejecutan en la medida en que son generadas. Con pruebas basadas en modelos fuera de línea (offline), los scripts generados pueden ser tomados por una

herramienta, por ejemplo, HP QuickTest, para administrar y ejecutar las pruebas.

Analizar los resultados de las pruebas: Para caso de prueba que reporte una falla, se debe determinar si es una falla en el sistema que estamos probando (SUT) ó en el modelo.

2. Contenido

Este capítulo trata de pruebas basadas en modelos, en este caso modelando el SUT con una máquina de estados finita (FSM) y luego utilizando ModelJUnit, obtendremos los grados de cobertura de las pruebas y la representación del modelo.

La organización de este capítulo es la siguiente: en el apartado 3.1 y 3.2 especificamos brevemente ModelJUnit. A continuación en el apartado 3.3 presentamos el caso de estudio, bajo el cual se presenta la aplicación y los resultados de este enfoque para pruebas. Finalmente, en las últimas secciones las conclusiones, hacia dónde dirigiremos nuestros esfuerzos y las referencias bibliográficas.

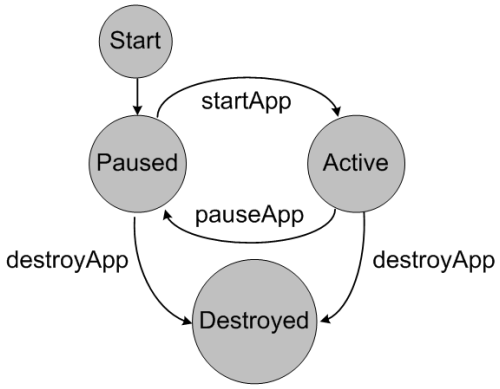
3. ModelJUnit

ModelJUnit es una librería de software libre desarrollada en java que extiende la funcionalidad de JUnit para soportar la ejecución de pruebas basadas en modelos. Su implementación está sustentada en el desarrollo del modelo del sistema a partir de máquinas de estados finitas que serán implementadas en Java.

Por otra parte, ModelJUnit presenta una serie de resultados basados en diferentes métricas de cubrimiento de las pruebas sobre el sistema de manera gráfica, ofreciendo un resumen que facilitará el análisis del mismo. [2]

3.1. FSM y ModelJUnit

El sistema será modelado por una máquina de estados finita (FSM), que representa el comportamiento del sistema. Una máquina de estados finita contiene tres elementos: los estados, representados en nodos; las transiciones del sistema de un estado a otro, representadas con los arcos dirigidos; las condiciones que deben ser cumplidas para efectuar las transiciones de estado, representadas por las etiquetas en los arcos. [3].



Gráfica 2. Máquina de estados finita. Tomado de Sun

Para este fin se tomarán los diferentes estados del sistema y se utilizarán guardas que permitirán especificar las características del sistema para pasar a un estado específico, permitiendo definir precondiciones, además se validará las características finales del estado aprovechando las características de validación a través de aserciones que ofrece JUnit, con lo que se definirán post-condiciones.

3.2. Pruebas unitarias para ModelJUnit

ModelJUnit amplía la funcionalidad de las pruebas unitarias de manera que se podrán ejecutar no solo validaciones sobre los datos de un procedimiento específico, sino que se podrán validar también secuencias de procedimientos analizando y validando los datos del sistema en diferentes estados y a través de diferentes caminos de ejecución, lo cual permitirá abarcar un mayor rango de pruebas.

Dentro del proceso de pruebas basadas en modelos, ModelJUNit, juega un rol en el paso 4: ejecución de los casos de pruebas concretos y una parte en el paso 5: análisis de los resultados de las pruebas, específicamente en lo que tiene que ver con el cubrimiento.

3.2. Caso de estudio

Se ha desarrollado a manera de caso de estudio una aplicación de un cajero automático que administrará información de manejo de transacciones básicas financieras (debitar y acreditar) sin mayores restricciones.

Paralelamente al desarrollo de la aplicación se crearán las pruebas unitarias usando JUnit, se ejecutarán las mismas sobre las clases controladoras del sistema y se tendrá un record de los resultados obtenidos.

Una vez efectuado este proceso se procederá a desarrollar un modelo del sistema para pruebas a través de una máquina de estados finita que será codificada en Java utilizando el API de ModelJUnit, y se ejecutarán estas pruebas manteniendo igualmente un récord de los resultados obtenidos.

Finalmente se procederá a comparar el cubrimiento y resultados de los dos esquemas de pruebas, además de la inversión de esfuerzo, resaltando las características específicas que ModelJUnit ha extendido y agregado a las conocidas anteriormente.

3.2.1. Características generales del caso de estudio

Se ha creado una entidad financiera que tiene cajeros automáticos distribuidos en el territorio. Esta entidad administra la información financiera de todos sus clientes a quienes les asigna una cuenta bancaria única.

El cliente podrá efectuar las operaciones de debitar y acreditar dinero de su cuenta a través del cajero obviando restricciones complejas como de longitudes de ingreso de datos, pero manteniendo las básicas, esto es: el saldo deberá estar entre 0 y 2147483647 pesos inclusive (un saldo negativo sería ilógico y el límite superior lo define el campo estático MAX_VALUE de la clase Integer).

Para efectuar cualquier operación el cliente deberá haberse identificado con su nombre y contraseña. Se asumirá un mismo nombre de entrada y no se harán pruebas sobre este campo debido a que esta información deberá ser leída a partir de una tarjeta y este procedimiento es ejecutado por un sistema electrónico externo al evaluado.

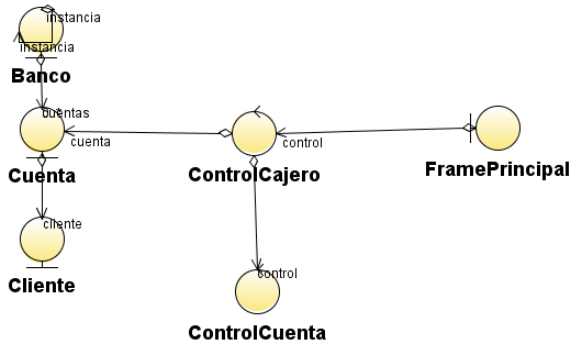
Cada cliente tendrá asignada una única contraseña que deberá introducir una vez leído el nombre, para esto tendrá tres oportunidades. En caso de introducir tres veces una contraseña errónea será notificado el cliente. Una vez ingresado al sistema, el cliente podrá efectuar cuantas transacciones desee sin abandonar la aplicación.

3.2.2. El desarrollo del caso de estudio

El caso de uso se ha desarrollado en Java, implementando el patrón Modelo Vista Controlador, de manera que las pruebas serán ejecutadas sobre las clases pertenecientes al módulo del controlador. Estas pruebas serán desarrolladas sobre JUnit 4.x y serán ejecutadas individualmente para comprobar el sistema.

El sistema no tendrá información persistente, de manera que al iniciar la aplicación se leerán datos básicos de clientes para las pruebas y los datos deberán ser consistentes con el flujo financiero que se efectúe mientras la aplicación esté en ejecución.

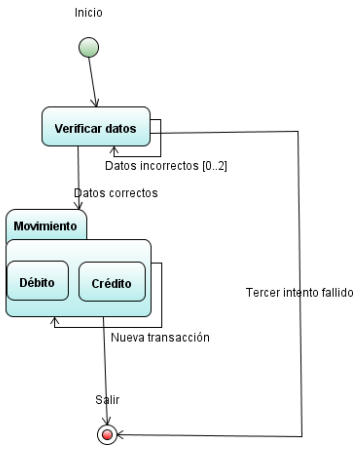
Para una mayor claridad y mejor desarrollo de la aplicación, se ha generado una arquitectura definiendo un modelo con un diagrama de clases UML.



Gráfica 3. Diagrama de clases del caso de estudio

Vemos acá el diagrama de clases, donde se pueden identificar claramente las clases de modelo, las de presentación y las de control (siendo estas últimas las que serán probadas).

También hemos generado un diagrama de Estados UML (que nos servirá también para desarrollar, a partir de él un modelo para pruebas definiendo una máquina de estados finita). Básicamente el diagrama de estados UML es una variante del diagrama de estados propuesto por Harel [4], quien extiende la funcionalidad de las máquinas de estado finita (EFSM), por ejemplo con los superestados, como se evidencia en el diagrama de la gráfica 4.



Gráfica 4. Diagrama de estados del caso de estudio

Tenemos definidos los posibles estados del sistema y qué acciones se deben seguir para ir de un estado a otro, por ejemplo, notamos que en el momento de verificar datos se podrá repetir esta operación un máximo de 2 veces, siendo la tercera la que obligará a abandonar la aplicación.

3.2.3. Datos de prueba del caso de estudio

Para nuestro sistema hemos establecido un usuario a probar cuyo nombre será "Miguel" y la contraseña "1234". Este usuario iniciará con una cantidad de 1.000.000.

Nombre	Contraseña	Salida esperada
Miguel	1234	Verdadero
Miguel Miguel	1233 1234	Falso Verdadero
Miguel Miguel Miguel	1233 1233 1233	Falso Falso TercerIntentoExcepcion

Tabla 1. Datos de prueba de "verificar datos"

Tenemos acá los tres casos distintivos de ingreso al sistema:
1. El caso en que el cliente ingresó la contraseña correcta.
2. El caso en que el cliente ingresa una contraseña incorrecta y luego la contraseña correcta.
3. El caso en que el cliente ingresa tres veces consecutivas una contraseña incorrecta.

Nótese que si deseamos probar el caso en que se ingrese dos veces la contraseña incorrecta y a la tercera oportunidad el ingreso correcto deberemos crear un nuevo grupo de datos de prueba y así para cada nuevo escenario que deseemos validar.

Tipo	Valor	Salida esperada
DEBITO CREDITO	500.000 1.500.000	Verdadero Verdadero
CREDITO	1.000.001	Falso
DEBITO CREDITO	500.000 1.500.001	Verdadero Falso
DEBITO	5.000.000	Verdadero

Tabla 1. Datos de prueba de "movimiento"

En este caso tenemos tres casos, verificaremos que no pueda retirar más del dinero que tiene disponible en la cuenta y que cuando ingrese dinero quede registrado y sea posible su retiro. Nuevamente nótese que si se desea verificar una serie de transacciones más compleja se tendrá que establecer un nuevo set de datos de pruebas.

3.2.4. Desarrollo y resultados de pruebas unitarias del caso de estudio

Se ha desarrollado el código para pruebas obteniendo casos como:

```
@Test
public void verificarDatosCorrectos(){
    String nombre = "Miguel";
    String contrasenia = "1234";
    assertTrue(instance.verificarDatos(nombre,
contrasenia));
}

@Test
public void testVerificarDatosDosVeces(){
    String nombre = "Miguel";
    String contrasenia = "1233";
    assertFalse(instance.verificarDatos(nombre,
contrasenia));
}

@Test
public void testVerificarDatosIncorrectos3() throws
Exception {
    String nombre = "Miguel";
    String contrasenia = "1233";

    assertFalse(instance.verificarDatos(nombre,
contrasenia));
    assertFalse(instance.verificarDatos(nombre,
contrasenia));

    try{
        assertFalse(instance.verificarDatos(nombre,
contrasenia));
        fail("debía lanzar una excepción de tercer
intento");
        }catch(TercerIntentoExcepcion e){}
    }
}
```

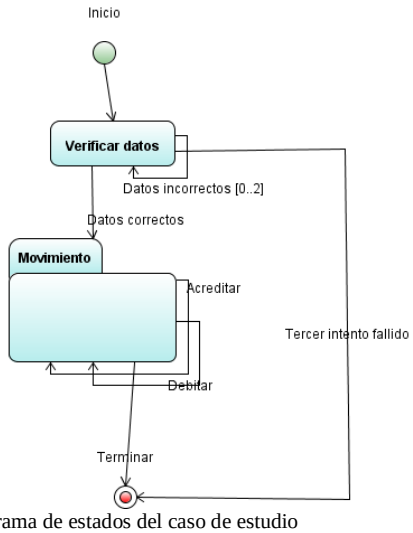
Se ejecutaron las pruebas unitarias obteniendo línea verde, sin encontrar errores en el sistema tanto en la

verificación de datos como en el movimiento transaccional.

3.2.5. Modelo para pruebas FSM del caso de estudio

Para aprovechar ModelJUnit deberemos entonces modelar el sistema para pruebas a través de una máquina de estados finita. Cabe destacar dos puntos importantes:

1. Los diagramas de estados UML son esencialmente máquinas de estados finitas.
2. El modelo del sistema para pruebas generalmente no es el mismo del modelo del sistema, ya que habrá datos, métodos y hasta clases que no serán relevantes al momento de probar la funcionalidad del sistema, así como también puede darse el caso en que uno de estos artefactos en el sistema cumpla con una funcionalidad que deba ser desglosada para una prueba más minuciosa.



Gráfica 5. Diagrama de estados del caso de estudio

En este diagrama podemos ver claramente que el caso de Debito y Crédito que habían sido modelados como estados del sistema se han convertido en acciones que mantendrán al sistema en el estado Movimiento.

Una vez desarrollado este diagrama, y soportándose en la documentación del API de ModelJUnit, se ha desarrollado la codificación de la máquina de estados finita que modelaría al sistema, la cual tiene por lo menos los siguientes métodos:

Object getState(): Este método retorna el estado actual de la EFSM

Void reset(boolean): Este método reestablece la ESFM a su estado inicial.

@Action void *name*(): La EFSM debe definir varias de estas anotaciones @Action. Estos métodos definen las transacciones de la ESFM.

boolean *name*μGuard(): Cada método de transición de estados puede opcionalmente tener una "guarda" o alerta.

Obteniendo, así, como resultado el siguiente código de ModelJUnit:

```
public class PruebasModel implements FsmModel {

  public enum Estados {VERIFICAR_DATOS, MOVIMIENTO, CANCELAR,
TERMINAR};

  private Estados estado;
  private ControlCajeroTest test;

  public PruebasModel() {
    test = new ControlCajeroTest();
    estado = Estados.VERIFICAR_DATOS;
  }

  public String getState() {
    return String.valueOf(estado);
  }

  public void reset(boolean testing) {
    test.reset();
    estado = Estados.VERIFICAR_DATOS;
  }

  public boolean verificarDatosCorrectosGuard() {
    return estado == Estados.VERIFICAR_DATOS;
  }

  @Action
  public void verificarDatosCorrectos() throws Exception {
    test.testVerificarDatosCorrectos();
    estado = Estados.MOVIMIENTO;
  }

  public boolean verificarDatosIncorrectosGuard() {
    return estado == Estados.VERIFICAR_DATOS;
  }

  @Action
  public void verificarDatosIncorrectos() throws Exception {
    test.testVerificarDatosIncorrectos();
    estado = Estados.VERIFICAR_DATOS;
  }

  public boolean acreditarGuard() {
    return estado == Estados.MOVIMIENTO;
  }

  @Action
  public void acreditar() throws Exception {
    test.testAcreaditar();
    estado = Estados.MOVIMIENTO;
  }

  public boolean debitarGuard() {
    return estado == Estados.MOVIMIENTO;
  }

  @Action
  public void debitar() throws Exception {
```

```
    test.testDebitar();
    estado = Estados.MOVIMIENTO;
  }

  public boolean cancelarGuard(){
    return estado == Estados.MOVIMIENTO;
  }

  @Action
  public void cancelar() {
    test.testCancelar();
    estado = Estados.VERIFICAR_DATOS;
  }

  public boolean salirGuard(){
    return estado == Estados.MOVIMIENTO;
  }

  @Action
  public void salir() {
    test.testCancelar();
    estado = Estados.TERMINAR;
  }

}
```

Donde se puede notar más claramente la manera en que podrían interactuar los estados y las guardas.

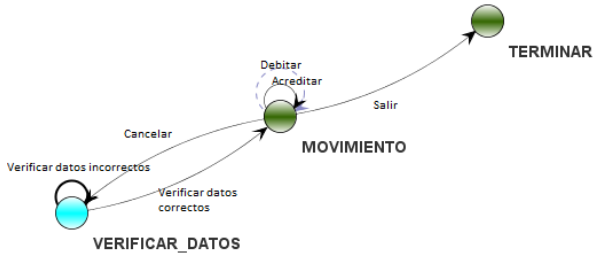
Cuando se terminó de modelar el sistema, se pudieron hacer algunos cambios a las pruebas unitarias, de manera que se podía "limpiar" el código, pues eran necesarias menos pruebas (algo como crear todos los colores a partir de los colores primarios), ya que la ejecución secuencial y aleatoria de los casos obviaba muchos escenarios analizados anteriormente. El nuevo código de pruebas quedó como sigue:

```
@Test
public void testVerificarDatosCorrectos() throws
Exception {
    String nombre = "Miguel";
    String contrasenia = "1234";
    cantIntentos=0;
    assertTrue(instance.verificarDatos(nombre,
contrasenia));
}

@Test
public void testVerificarDatosIncorrectos() throws
Exception {
    String nombre = "Miguel";
    String contrasenia = "1233";
    cantIntentos++;
    if (cantIntentos<3){
        assertFalse(instance.verificarDatos(nombre,
contrasenia));
    }else{
        try{
            assertFalse(
                instance.verificarDatos(
                                nombre,
                                contrasenia));
            fail("debía lanzar una excepción de
tercer intento");
            }catch(TercerIntentoExcepcion e){}
        }
    }
```

Con lo que, a partir de dos métodos de pruebas se podría lograr una amplia gama de ejecución a probar.

ModelJUnit tiene una herramienta gráfica que está en desarrollo y con la que se puede ver el modelo que se ha creado como una máquina de estados finita, en nuestro caso es:



Gráfica 6. FSM del caso de estudio

Se puede comprar con el diagrama de estados del caso de estudio (gráfica 5) con lo que se notarán las similitudes, de manera que cada estado se convierte en un nodo y cada acción en un arco.

3.2.6. Resultados de pruebas con ModelJUnit del caso de estudio

Con ModelJUnit se pueden ejecutar las pruebas estableciendo diferentes parámetros, como el algoritmo de ejecución de pruebas, la cantidad de pruebas a ejecutar, entre otros. Cuando se ejecutaron las pruebas por primera vez, notamos que existía un problema en nuestro diseño, ya que, al no manejar datos persistentes, nuestro sistema bancario perdía la información en el momento en que el cliente deseaba reingresar al sistema, esto se notó al ejecutar transacciones en secuencia, y se solucionó implementando el patrón singleton para mantener una única instancia de la entidad Banco.

Una vez solucionado este problema se escogió el algoritmo "random walk" con 10, 20 y 50 pruebas, obteniendo como resultado:

- Longitud de Pruebas: 10
  - State coverage = 2/3
  - Transition coverage = 4/6
  - Transition pair coverage = 6/16
  - Action coverage = 4/6
- Longitud de Pruebas: 20
  - State coverage = 2/3
  - Transition coverage = 5/6
  - Transition pair coverage = 9/16

- Action coverage = 5/6
- Longitud de Pruebas: 50
  - State coverage = 3/3
  - Transition coverage = 6/6
  - Transition pair coverage = 12/16
  - Action coverage = 6/6

Pasamos a explicar cada una de las métricas anteriormente generadas:

State coverage: El número de estados del modelo que son visitados. Por ej: (2/3)
- VERIFICAR_DATOS
- MOVIMIENTO

Transition coverage: El número de transiciones de estado que son recorridas. Por ej.: (4/6)
- verificar datos correctos
- verificar datos incorrectos
- debitar
- acreditar

Transition pair coverage: El número de transiciones adyacentes que son recorridas. Por ej:(6/16)
- verificar datos correctos / debitar
- verificar datos correctos / acreditar
- debitar / salir
- acreditar / salir
- debitar / cancelar
- acreditar / cancelar

Action coverage [5]: El número de transiciones de estado que son recorridas sin tener en cuenta el estado inicial. Por ej.: (4/6)
- verificar datos correctos
- verificar datos incorrectos
- debitar
- acreditar

Esta información la ofrece la librería para su análisis, sin embargo, cabe destacar que la diferencia de tiempo de ejecución para este sistema (que evidentemente es sumamente sencillo) era casi exponencial, por lo que se necesitará lograr un buen análisis previo de la cantidad de pruebas a ejecutar en una relación costo beneficio.

4. Conclusiones

- El desarrollo de las pruebas se facilita al no necesitar crear una prueba independiente para cada secuencia posible
- El cubrimiento de secuencias es mayor supliendo algunos errores humanos
- La herramienta gráfica tiene muchas falencias
- A pesar de poder partir de un diagrama, la herramienta no tiene la capacidad de generar el código de la FSM automáticamente

5. Trabajo Futuro

- Integrar UML-Based Statistical Test Case Generation con ModelJUnit para optimizar la ejecución de pruebas
- Desarrollar una herramienta gráfica basada en ModelJUnit para la creación de FSM para ejecución de pruebas
- Investigar notación Pre/Post (específicamente OCL) para creación de pruebas a partir de modelos

6. Referencias

[1] Mark Utting Et Al, The ModelJUnit Model-Based Testing Tool, http://www.cs.waikato.ac.nz/~marku/mbt/modeljunit/
[2] Mark Utting and B.Legeard, “Practical Model-Based Testing”, Morgan Kaufmann Publishers. 2007.
[3] Brian Schwab, “AI Game Engine Programming”, Cengage Learning, 2009
[4] David Harel, “Statecharts: A Visual Formalism for Complex Systems”, 1987
[5] Jonathan Jacky Et Al, “Model-Based Software Testing and Analysis with C#”, 2007