ANÁLISIS Y COMPARACIÓN DE LAS PROPUESTAS RECIENTES PARA DISEÑAR CASOS DE PRUEBA DESDE LOS CASOS DE USO ORIENTADOS A VERIFICAR LOS ASPECTOS FUNCIONALES DEL SOFTWARE

Edgar Serna M.

Trabajo de tesis para optar al título de Magister en Ingeniería de Sistemas

UNIVERSIDAD NACIONAL DE COLOMBIA MEDELLÍN

FACULTAD DE MINAS

ESCUELA DE SISTEMAS

Director
Dr. FERNANDO ARANGO ISAZA

MEDELLÍN - ANTIOQUIA

2011

RESUMEN

Este documento contiene la relación del trabajo de investigación de la tesis para optar al título de maestría en ingeniería de software, respecto de un análisis y comparación de las propuestas recientes para el diseño de casos de prueba desde los casos de uso orientados a verificar los aspectos funcionales del software.

En el transcurso de la investigación consultamos la literatura hasta encontrar las propuestas promulgadas de 2000 en adelante alrededor de esta temática. Las propuestas seleccionadas se aplicaron a un estudio de caso con el objetivo de analizar y evaluar los resultados a través de métricas valorativas. Posteriormente, se identificaron los conceptos que las soportan y finalmente se estructuró una nueva propuesta, con el objetivo de recoger las mejores prácticas que exponen y con base en estos conceptos. También se presenta una descripción de cada una de las propuestas y al final una identificación de los conceptos que las soportan.

Este trabajo parte de una necesidad en la industria y la academia relacionada con las pruebas del software. La industria requiere propuestas que describan metodologías para aplicar pruebas desde las fases iniciales del ciclo de vida del software, y la academia necesita textos que describan métodos útiles o de fácil incorporación a los procesos formativos. Por lo que esta tesis es una fuente de consulta para la industria, dado su carácter empírico, y para la academia, dado su contenido teórico-aplicativo.

La consulta del estado del arte permitió identificar nueve propuestas que describen métodos para diseñar casos de prueba desde los casos de uso para verificar los aspectos funcionales del software. Cada una de ellas detalla una serie de pasos para lograr el objetivo planteado, y aunque al aplicarlos sobre el estudio de caso se encontraron dificultades también se identificaron buenas prácticas. El resultado del proceso se estructuró en tablas comparativas, las cuales constituyeron la fuente para identificar los conceptos y procesos clave de las propuestas que luego se estructuraron en un método que recoge las mejores prácticas.

Al final de este documento se plantean cuestiones y temas para desarrollar trabajos futuros alrededor de esta temática que no se cubren en el desarrollo de la tesis dado que están por fuera de su alcance. Entre estos temas cabe mencionar: pruebas estructurales o de caja blanca, automatización de las pruebas, validación del software, requisitos no funcionales, entre otros.

Este trabajo hace dos aportes fundamentales: 1) el análisis comparativo, que es novedoso dado que en la revisión a la literatura no se encontraron estudios similares; y 2) la integración, estructuración, descripción y aplicación de una nueva propuesta para diseñar casos de prueba desde los casos uso, en la que se utiliza las buenas prácticas de las propuestas analizadas, y se aportan conceptos novedosos y actuales en el campo de la ingeniería del software.

TABLA DE CONTENIDO

LISTA DE TABLAS

LISTA DE FIGURAS

INTRODUCCIÓN

En esta tesis se aborda la temática de la estructuración de casos de prueba desde los casos de uso para verificar los aspectos funcionales del software. Es un trabajo que tiene como objetivo comparar las propuestas recientes, relacionadas con esta temática, mediante una aplicación empírica a un estudio de caso, para analizarlas e identificar los conceptos y métodos de cada una de ellas; posteriormente estructurar, de acuerdo con los resultados, una nueva propuesta que reúna las mejores prácticas de estos trabajos, y a la que se incorporan conceptos e ideas novedosas como nuevo conocimiento para discusión en la comunidad.

Primero se hizo una amplia consulta del estado del arte con el objetivo de encontrar: 1) los trabajos que han desarrollado estudios analítico-comparativos semejantes al que aquí se plantea, y 2) las propuestas para estructurar casos de prueba desde los casos de uso, que se hayan promulgado recientemente –2000 en adelante. Esta información constituyó la base para aplicar cada uno de los pasos propuestos en el proyecto. En un marco ontológico unificado se describen los objetivos, conceptos y métodos de los enfoques del estado del arte, para determinar los procesos sugeridos en las propuestas y aplicarlos al estudio de caso. Luego, se aplicaron las propuestas seleccionadas a un estudio de caso para verificar la eficiencia del conjunto de casos de prueba generado a partir de los casos de uso en la verificación de los aspectos funcionales del software. Posteriormente, se definieron los criterios para determinar la eficiencia del conjunto de casos de prueba que fueron generados en la aplicación de las propuestas al estudio de caso; se tabularon y compararon los resultados de la aplicación de las propuestas, identificando su contribución a los objetivos, conceptos y métodos del estado del arte. Finalmente, se analizaron estos resultados con el objetivo de determinar las mejores prácticas, y se desarrolló una nueva propuesta para estructurar casos de prueba desde los casos de uso para verificar los aspectos funcionales del software.

La motivación para trabajar en esta área surgió luego de realizar el proyecto “Estructuración de una metodología genérica para realizar pruebas de caja negra en los sistemas de información”, que fue patrocinado por la Fundación Universitaria Luis Amigó y la industria local de desarrollo de software. El objetivo de ese proyecto fue estructurar una metodología genérica para aplicar pruebas de caja negra, que se logró suficientemente y que quedó registrada en un libro que se encuentra en edición. Al terminar la primera fase del proyecto, el estudio del estado del arte, se encontraron tres áreas en pruebas del software a las que les faltaba más trabajo: 1) formalización de la especificación de requisitos; 2) automatización de la aplicación del conjunto de casos de prueba; y 3) generación de casos de prueba desde los casos de uso. Este hecho generó tres nuevos proyectos de investigación, uno de los cuales se tomó para ejecutar en el desarrollo de esta tesis de maestría.

Aunque la temática de las pruebas del software es bastante amplía, este trabajo se centra en la generación de casos de prueba desde los casos de uso. Esta sub-área temática se refiere a que, para cumplir el objetivo de las pruebas, es necesario estructurar los casos de prueba desde el momento mismo en que se elicitan los requisitos y las reglas del negocio. Es decir, pensar en las pruebas desde el momento mismo en que se analiza el problema a resolver. Esto se logra cuando la especificación en casos de uso se utiliza para estructurar el conjunto de casos de prueba, y se aplica inmediatamente para verificar los aspectos funcionales del producto que se está desarrollando. Lo que difiere de la ingeniería tradicional en la que las pruebas se piensan y ejecutan como una fase al final del ciclo de vida.

En el estado del arte se encontró una amplia variedad de definiciones acerca de lo que son los casos de prueba (IEEE, 1983) (IEEE, 1990) (Beizer, 1995) (Binder, 2000) (Patton, 2000), pero en este trabajo se asume que un caso de prueba es una pregunta que se le hace al programa, ya que el objetivo de aplicar la prueba es obtener información. Una implicación importante de esta definición es que un caso de prueba debe ser razonablemente capaz de revelar información, por lo que se convierte en un conjunto de condiciones o variables de acuerdo con las cuales un probador puede determinar si una aplicación o producto software funciona correctamente o no.

El conjunto de casos de prueba se convierte entonces en uno de los elementos más importantes de la prueba, y debido a que existen diferentes enfoques para diseñarlo, cada propuesta de prueba describe estrategias propias para diseñarlo y aplicarlo. El otro elemento fundamental de la prueba es la valoración de los casos de prueba, y dada la variedad de propuestas existentes y a que de su calidad depende la efectividad misma de la prueba, es necesario conocer las características de cada una para determinar la efectividad de los casos de prueba para detectar anomalías. Diseñar y valorar casos de prueba, lo mismo que desarrollar software, es una actividad que requiere de inteligencia y creatividad, por lo que necesariamente la realizan personas con alto conocimiento técnico de las pruebas y del dominio del producto software objetivo. La valoración es un proceso estratégico para verificar que el conjunto de casos de prueba realmente ofrece la información que se especificó y encuentra las anomalías que el producto tiene. Este proceso determina la efectividad final de la prueba y la calidad del producto que se construye.

La prueba misma, aunque aparentemente creativa y completa, no puede garantizar la ausencia de anomalías en el software y, más allá de las cuestiones psicológicas alrededor de ella, la consideración más importante es poder estructurar, diseñar y aplicar eficientemente un conjunto de casos de prueba. Este diseño es importante debido a que la prueba de cobertura completa es imposible, por lo que la estrategia obvia es tratar de hacerlo tan completo y eficiente como sea posible. Teniendo en cuenta las limitaciones de tiempo y de costos, la cuestión clave de la prueba se convierte en *qué parte de este conjunto tiene la mayor probabilidad de detectar la mayoría de los errores en el menor tiempo y al menor costo*.

Cuando las empresas de desarrollo implementan las pruebas de software pueden identificar las anomalías en sus productos antes que lo hagan sus clientes. Esto es vital ya que les permite mantener la credibilidad –algo que tiene mucho más valor que el costo mismo de las pruebas. Contrariamente a la creencia popular, el principal objetivo de las pruebas de software no es demostrar que el sistema funciona o incluso encontrar errores, sino más bien, que es reducir el trabajo redundante. Sólo a través de la reducción de procesos es que se obtienen los verdaderos beneficios de las pruebas de software –por ejemplo, el ahorro de costos y tiempo durante el desarrollo. Por todo esto, la importancia de las pruebas de software y su impacto no pueden ser subestimados. Las pruebas de software es un componente fundamental de la garantía de la calidad del software y representa una revisión de la especificación, el diseño y la codificación. La mayor visibilidad de los sistemas software y el costo asociado con los errores del software son factores que motivan a planificar el software incluyendo las pruebas desde las fases iniciales.

El objetivo de las pruebas, sin importar cómo se realicen, es asegurar desde las primeras fases del ciclo de vida que el producto no fallará una vez esté en funcionamiento, que respetará las reglas del negocio y que llevará a cabo lo que fue especificado. Pero, aunque el costo de corregir un error en las fases finales del desarrollo es mucho más alto que hacerlo en las fases iniciales, la mayoría de los productos de la Ingeniería de Software tradicional se desarrollan y comercializan

sin tener esto en cuenta, por lo que el concepto de pruebas, diseñadas y aplicadas eficientemente en todo el ciclo de vida del producto, parece no ser atendido por todos.

Es por esto que es necesario desarrollar una serie de procesos de investigación que permitan seleccionar lo más inteligentemente posible los datos para la prueba y los casos de prueba asociados; por lo que realizar un análisis a las propuestas existentes para diseñar casos de prueba, como el que aquí se presenta, es en sí una estrategia acertada de prueba, ya que en el proceso es posible generar una combinación de elementos que incremente la eficiencia y mejore la cobertura de los casos de prueba estructurados. Es decir, un trabajo de este tipo puede generar un enfoque razonablemente riguroso de diseño de casos de prueba desde los casos de uso que complemente los estudios existentes, y mejore los procesos para aplicar las pruebas.

En el estado del arte de las pruebas de software se encontraron trabajos que aportan ideas generales acerca del área. Algunos de ellos son descripciones genéricas de las concepciones teóricas, otros son aplicaciones de metodologías de pruebas, y otros se centran en describir taxonomías de las pruebas y las técnicas de prueba. Entre estos trabajos se encuentra el de Beizer (1990), que ofrece una visión amplia y completa de las pruebas de software y hace hincapié en los modelos formales para la prueba. Presenta una visión general del proceso de las pruebas y las razones y objetivos para ejecutarlo; distingue cuidadosamente entre las pruebas y la depuración; ofrece una caracterización de las pruebas en funcionales y estructurales, e identifica y caracteriza claramente los diferentes tipos de pruebas que se presentan en las empresas de desarrollo. Es un texto altamente académico y teórico de mucha utilidad para los cursos en esta área, pero que no llega a tratar casos de estudio de forma aplicativa. Como un especialista en el campo de pruebas de software, Beizer utiliza en su publicación de 1995 una serie de ejemplos de pruebas de formas de impuestos del IRS –Internal Revenue Service– para demostrar cómo utilizar una gran cantidad de técnicas de prueba de caja negra probadas y aceptadas para validar los requisitos de las formas a las que se refiere el software. Introduce todo tipo de métodos y diseña las pruebas de acuerdo con el modelo que han depurado los lectores de su libro anterior; además, incluye numerosas pruebas de autoevaluación, como un aporte novedoso en su momento.

Testing Computer Software (Kaner et al., 1999) es uno de esos libros raros que aborda el tema de los problemas del Ingeniero de verificación. Actualmente, la mayoría de los libros en pruebas de software están dirigidos a los desarrolladores e incluso muchos de ellos no están escritos apropiadamente o tratan de cubrir demasiados temas del área. Este libro, sin embargo, aunque amplio en su alcance, hace un buen trabajo al tratar todas las áreas importantes en verificación y pruebas. Por su parte, el libro de Lewis (2000) se enfoca en ayudar a incrementar la calidad en el desarrollo de software y a mejorar continuamente cada uno de sus procesos. Primero presenta una visión general de los principios básicos de calidad y cómo se puede aplicar en el ciclo de mejora continua de las pruebas de software; luego revisa las pruebas en el ciclo de vida cascada, seguido de una metodología de pruebas exhaustivas en desarrollo rápido de aplicaciones cliente/servidor e Internet. En la parte final proporciona un análisis completo a las herramientas de modernas de prueba. Es un libro académico de fácil consulta y acceso para comprender los procesos de prueba aplicados en los procesos de la ingeniería de software.

Con base en las necesidades de la comunidad educativa, y de los profesionales de software, el libro de Burnstein (2003) tiene un enfoque único para la formación en pruebas de software. Introduce los conceptos de pruebas utilizando el Modelo de Madurez de Pruebas –TMM– como marco de referencia. Promueve el crecimiento y el valor de las pruebas de software como una profesión, y presenta los aspectos técnicos y de gestión de las pruebas de forma clara y precisa.

Además, insiste en la necesidad de fomentar el surgimiento de cursos de postgrado que cubran estas temáticas, y describe el desarrollo de software como una disciplina de la ingeniería. Por su parte, Myers (2004) proporciona un análisis más práctico que teórico de la finalidad y la naturaleza de las pruebas de software. Hace hincapié en las metodologías efectivas para el diseño de casos de prueba. Recorre de forma exhaustiva los principios psicológicos y económicos, los aspectos de gestión de pruebas, las herramientas de prueba, las pruebas de orden superior, y las inspecciones de código y de depuración, y presenta una extensa bibliografía. Por todo esto, los desarrolladores, en todos los niveles, y los estudiantes de programas en ciencias computacionales, podrán encontrar en esta obra una buena referencia.

De estos aportes es posible concluir que el área de las pruebas del software se estructura de acuerdo con el contenido de la Fig. 1.

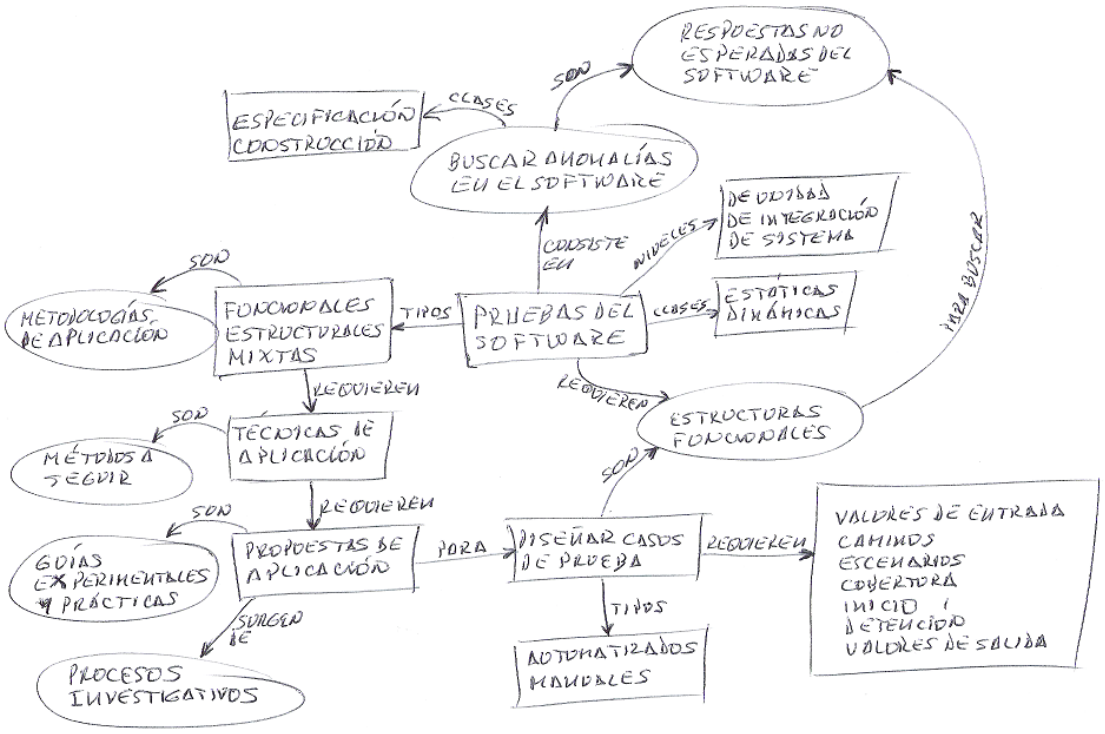

Fig. 1 Mapa de las pruebas del software

Las pruebas del software se clasifican en tres tipos generales: funcionales, estructurales y funcionales/estructurales. Las pruebas funcionales, también conocidas de caja negra, basa la estructuración de los casos de prueba en las especificaciones del software bajo prueba. Las funciones del programa se prueban aplicando valores de entrada y comparándolos con los valores de salida, y no considera su estructura interna. Mientras que las pruebas estructurales, o de caja blanca, es una forma de probar el software conociendo el funcionamiento y la lógica internos del código. Las pruebas funcionales/estructurales, también conocidas como caja gris, es un proceso de prueba en el que se utiliza tanto procesos funcionales como estructurales. Para cada uno de estos tipos de pruebas se han desarrollado técnicas que describen metodologías cuya aplicación determina el nivel de prueba seleccionado, es decir, pruebas de unidad, de integración y de sistema. Las de unidad, seleccionan porciones individuales del programa para aplicar las pruebas; las de integración, reúnen las porciones individuales anteriormente probadas y las integran en módulos para realizarles pruebas; y las de sistema, se aplican a todo el programa luego de las integraciones previas.

Para cada uno de estos tipos y niveles de prueba se han propuesto métodos o técnicas de aplicación, las cuales describen pasos que los equipos de probadores siguen hasta lograr los objetivos planteados en el proceso de la prueba. Estas técnicas se encargan de proporcionar las directrices sistemáticas y operativas para estructurar y aplicar los casos de prueba, de acuerdo con la prueba seleccionada. Ejemplos de técnicas de prueba son las tablas de decisión, la partición de equivalencia, las pruebas de valor límite, las máquinas de estado finito, las pruebas basadas en la especificación formal, las pruebas aleatorias, las de casos de prueba desde los casos de uso, entre otras. A su vez, para cada técnica de prueba surgieron propuestas de aplicación para llevarlas a la práctica y lograr la detección de las anomalías. Una propuesta resulta de procesos de investigación que buscan la mejor forma de aplicar una técnica determinada. Ahora bien, existen dos clases de pruebas, las estáticas y las dinámicas, que determinan la forma en que se aplica el proceso de prueba al software. Si el equipo toma la decisión de probar el software de forma que mediante un seguimiento lógico al programa pueda determinar su calidad sin correrlo, entonces aplica una prueba estática; si por el contrario, decide ejecutar el programa, sea mediante animación o simulación, entonces ejecuta una prueba dinámica.

Para el trabajo que se describe en este texto se seleccionó el siguiente árbol de las pruebas:
Tipo: Funcionales
Nivel: Unidad
Clase: Dinámicas
Técnica: Casos de prueba desde los casos de uso

Se parte de un concepto de valor descendente para especificar el proceso de las pruebas: el nivel superior de un plan de pruebas es determinar qué tipo de prueba se desea iniciar, en este caso se trata de seleccionar entre pruebas funcionales, pruebas estructurales o pruebas mixtas; el sub-siguiente nivel es seleccionar la técnica que se quiere aplicar, para lo cual existe un significativo número para cada tipo de prueba; posteriormente, investigar cuáles son las propuestas para aplicar dicha técnica, para lo cual existen trabajos de investigación que dan soporte para su adecuada selección; y finalmente hacer un análisis para determinar las fortalezas y deficiencias de las propuestas seleccionadas y estructurar una propuesta como resultado de todo el proceso. Para la presente investigación se seleccionó: 1) *pruebas funcionales*, como el tipo de prueba a trabajar, 2) *casos de prueba desde los casos de uso* como la técnica a aplicar, y 3) *nueve propuestas* desde la literatura para analizarlas y evaluarlas. En esta selección, las propuestas debían cumplir con las siguientes características: *basarse en la técnica seleccionada, que fueran de reciente promulgación* –2000 en adelante–, y *partir de casos de uso descritos en lenguaje natural*. Estas características se seleccionaron teniendo en cuenta que el nivel de un proyecto de maestría debe ser suficiente para determinar adecuadamente el estado del arte acerca de un área específica del conocimiento, y que debe servir como base para la realización de un trabajo posterior en un nivel de mayor exigencia, como un doctorado.

También en el estado del arte, se encontró una serie de trabajos que realizan análisis comparativos semejantes al que se propone desde esta tesis. Entre ellos cabe mencionar trabajos como el de Basili & Selby (1987), quienes aplicaron una metodología experimental para comparar el estado de la práctica con tres técnicas de pruebas de software. Dicho estudio compara las estrategias de las pruebas en tres aspectos: la eficiencia para detectar errores, el costo de la detección, y la clase de errores encontrados, pero no presenta una alternativa que solucione o modifique los resultados. Kobrosly & Vassiliadiset (1988) analizan varios artículos técnicos en pruebas de software y proporcionan una muestra representativa de técnicas de pruebas funcionales, pero su estudio culmina con la presentación de una tabla comparativa.

En su investigación, Denger & Medina (2003) presentaron los resultados de una búsqueda en la literatura con el objetivo de identificar el estado actual de las técnicas en la generación de casos de prueba basados en los requisitos de usuario, pero no culminaron esa búsqueda con un análisis o comparación a dichas técnicas. Wagner (2004) compara la eficiencia de algunas técnicas para detectar anomalías en el software. Utiliza métricas basadas en la cobertura de código y un contador de errores, e introduce la métrica de intensidad de la falla orientada a la fiabilidad del software, pero le queda faltando la estructuración en una nueva propuesta de los resultados alcanzados. Igualmente, Seo & Choi (2006) presentan los resultados de su comparación empírica a cinco métodos de prueba funcional y detallan los diferentes resultados de aplicación, luego de ponerlas en práctica en un laboratorio sobre un sistema de software; su estudio se limitó a reportar resultados. Por esta misma línea, Gutiérrez *et al.* (2006) presentan el resultado de un estudio empírico a trece propuestas que diseñan casos de prueba desde los requisitos funcionales, pero su análisis comparativo no culminó con un artículo de difusión o con una nueva propuesta aplicativa. Finalmente, Allott (2007) concluye que las técnicas de prueba se enseñan en las universidades y en muchos centros de capacitación industrial, pero que en el mundo corporativo los probadores tienen dificultades para aplicarlas, por lo que deben recurrir a su conocimiento del dominio y a su experiencia para diseñar los casos de prueba.

Debido a los diferentes faltantes en el área de investigación, antes mencionados, es que surge la realización de este proyecto, ya que si antes de estructurar las pruebas es posible conocer las pautas generales para diseñar y estructurar un conjunto de casos de prueba para aplicar en un producto software, que garantice ser el más eficiente del universo resultante, sería posible entregarle al cliente un sistema con la mínima cantidad de errores posible. Los beneficios de este tipo de entregas repercutirían en el área de desarrollo de la organización, la autoestima del equipo de trabajo, la imagen de la empresa y el incremento de utilidades; el departamento de proyectos podría cumplir los plazos de entrega, disminuir costos, incrementar calidad, pasar del desarrollo a la puesta en producción con mayor eficiencia y efectividad; los proyectos generarían versiones de los producto más estables, y sería posible ofrecer mejor respuesta a cuestiones funcionales y aspectos técnicos; finalmente, repercutiría también en la satisfacción del cliente, lo cual incrementaría la confianza de su organización para explotar el nuevo sistema.

En la realización de este proyecto se responde a los siguientes interrogantes:

1. ¿Qué estudios han presentado propuestas recientemente para diseñar el conjunto de casos de prueba desde los casos de uso para las pruebas funcionales y como se relacionan entre sí?

2. ¿Qué tan eficaces son los casos de prueba diseñados con estas propuestas?

3. ¿Es posible diseñar una nueva propuesta para diseñar casos de prueba desde los casos de uso para pruebas funcionales a partir de un análisis comparativo a estos estudios?

Para responder a estos interrogantes se diseñaron las siguientes estrategias:

1. Describir en un marco ontológico unificado los objetivos, conceptos y métodos de los enfoques del estado del arte.

2. Definir los criterios para determinar la eficiencia del conjunto de casos de prueba que genera la aplicación de las propuestas seleccionadas para verificar los aspectos funcionales del software.

3. Aplicar las propuestas analizadas a un estudio de caso para verificar la eficiencia del conjunto de casos de prueba generado.

4. Comparar las propuestas identificando su contribución a los objetivos, conceptos y métodos del estado del arte.

En la realización del proceso de investigación se logró cubrir los objetivos totalmente, lo mismo que temáticas como las pruebas funcionales, la verificación del software, la experimentación, el diseño de casos de prueba, la estructuración de escenarios de prueba, el diseño de grafos de caminos, y la especificación de requisitos; pero no se estudiaron algunas temáticas que estaban por fuera de los alcances propuestos, como las pruebas estructurales, la automatización de las pruebas, la validación del software, la formalización de requisitos, entre otras.

El documento se encuentra estructurado de la siguiente forma: el capítulo I contiene los planteamientos iniciales de la tesis en los que se describe el problema de investigación, la propuesta metodológica y los antecedentes; el capítulo II describe el marco teórico de la investigación; en el capítulo III se describe la revisión a la literatura, que corresponde a las nueve propuestas seleccionadas para el análisis, con el objetivo de hacer un marco conceptual que permita reconocer el aporte del trabajo que aquí se presenta; el capítulo IV detalla los resultados generales del análisis a objetivos, conceptos y métodos aplicados, y se presentan conclusiones generales esos análisis; en el capítulo V se describe una propuesta para estructurar casos de prueba que resume las mejores prácticas de las propuestas analizadas, y se recomiendan investigaciones futuras. Los anexos I al III contienen tablas con información de las técnicas de pruebas funcionales, y el Anexo IV el detalle de la aplicación de la propuesta al estudio de caso de un ATM.

I. PLANTEAMIENTOS INICIALES

1. EL PROBLEMA DE INVESTIGACIÓN

Como se mencionó en la introducción, este trabajo se inscribe en el desarrollo del proyecto de investigación "*Diseño de una metodología genérica para la realización de pruebas de caja negra en los sistemas de información*". Se espera que la realización de esta tesis ayude a potencializar los objetivos planteados para ese proyecto:

- Crear comunidad regional alrededor de las pruebas del software.
- Determinar cuáles son las técnicas más utilizadas para aplicar pruebas funcionales en la industria local.
- Encontrar y clasificar los programas de Ingeniería en las universidades regionales en cuyos contenidos contemplen temas como calidad y pruebas del software, métodos formales, diseño de casos de prueba, entre otros.
- Tomar los resultados de este proceso investigativo como punto de partida para la publicación de una serie de artículos alrededor de la temática de las pruebas del software, y como base para iniciar otros proyectos de investigación.

1.1 Descripción del problema

El software no es diferente a otros procesos físicos en los que se recibe insumos y se genera productos, es diferente en la forma en que falla. La mayoría de los sistemas físicos fallan en un fijo y razonablemente pequeño conjunto de formas, pero el software puede fallar en muchas y extrañas formas y la detección completa de esas fallas es en general inviable. A diferencia de casi todos los sistemas físicos, la mayor parte de las fallas en el software son errores de diseño, no de fabricación: no sufre corrosión, desgaste o vencimiento, y por lo general no cambia hasta que se actualiza o se vuelva obsoleto, por lo que una vez que se entrega sus defectos serán enterrados de forma latente hasta su activación.

Existen varias definiciones acerca de lo que es la prueba del software:

- Es una actividad dirigida a la evaluación de un atributo o capacidad de un programa o sistema para determinar que cumple sus especificaciones (Hetzel, 1998).
- Es un proceso en el que se corre un programa o sistema con la intención de encontrarle errores (Myers, 2004).
- Consiste en verificar dinámicamente la conducta del programa bajo un conjunto finito de casos de prueba, y comparar los resultados con lo que se esperaba (Boehm, 1973).
- Es una actividad cuya realización consume al menos la mitad de la mano de obra utilizada para desarrollar un producto software (Brown & Lipow, 1973) (Wolverton, 1974) (Horowitz, 1975) (Goodenough, 1979) (Radatz, 1981).

Es decir, la prueba de software es un proceso planificado que tiene como objetivo encontrar la mayor cantidad de errores posible en un sistema antes que se entregue al cliente.

Existen varios tipos de pruebas, entre las que destacamos las pruebas funcionales y que se basan en la ejecución, revisión y retroalimentación de los requisitos funcionales, diseñados previamente para el software. El proceso de ejecución de estas pruebas consiste en estructurar, diseñar y aplicar un conjunto de casos de prueba con el objetivo de evaluar cada una de las opciones con las que cuenta el paquete informático. Su objetivo es verificar si el comportamiento que presenta el software cumple o no con lo esperado de acuerdo con su especificación, para lo que es necesario tener en cuenta el punto de vista del usuario (Beizer, 1990). Las funciones se prueban

con valores de entrada y luego se examinan las salidas, la estructura interna del programa raramente se considera (Kaner *et al*., 1999). El enfoque habitual para la detección de errores en este tipo de pruebas es que el probador selecciona un conjunto de datos de entrada y a continuación ejecuta el software con esos datos bajo unas condiciones particulares; para decidir si el software pasa o no la prueba necesita conocer, dados los valores de entrada y las condiciones de ejecución –escenario–, cuáles serán las salidas correctas esperadas del programa. Con esa información diseña e conjunto de casos de prueba.

El conjunto de casos de prueba es una serie de condiciones y valores con los cuales se ejercita el programa para analizar si los resultados que presenta validad o niegan los valores y comportamiento esperados, y se convierte en el elemento más importante de la prueba de un producto. Debido a que existen diferentes enfoques para diseñarlo, cada tipo de prueba propone estrategias propias para diseñarlo, aplicarlo y evaluarlo. Además, dada la variedad de propuestas existentes y a que de su calidad depende la efectividad misma de la prueba, es necesario conocer las características de cada una para determinar su efectividad al momento de detectar anomalías. Diseñar casos de prueba, lo mismo que desarrollar software, es una actividad que requiere de inteligencia y creatividad, por lo que necesariamente la realizan personas con alto conocimiento técnico de la prueba y del dominio del producto software objetivo.

Existen numerosas herramientas para automatizar la aplicación de las pruebas, pero muchas veces el diseño y gestión del conjunto de casos de prueba se realiza manualmente, y se aplica como una fase más del ciclo de vida del producto aislada de las demás actividades del proceso de desarrollo. La mayoría de las propuestas encontradas registran los casos de prueba en formatos poco estructurados, lo que impide que la evaluación del conjunto se sistematice y a que no tenga un cuerpo de conocimiento que guarde los datos acerca de su efectividad, nivel de cobertura y posible replicación (Baudry *et al*., 2002).

Es necesario tener claros los objetivos de la prueba para determinar la efectividad de un conjunto de casos de prueba; no se trata sólo de observar un conjunto de ejecuciones del programa por el simple hecho de revisarlo, debe hacerse con objetivos claros y metas por cumplir, ya que realizar pruebas es ejecutar los componentes del programa antes de entregarlo al cliente. Mientras más claridad se tenga acerca de la efectividad de los casos de prueba seleccionados más efectivas serán las pruebas aplicadas. Es por esto que la realización de esta investigación es importante dado su potencial aporte al proceso de estructuración, diseño y evaluación del conjunto de casos de prueba para mejorar la calidad de los productos software; y a que es un campo que se ha trabajado de tan diversas formas que hace falta un análisis que pueda conducir a su unificación.

1.2 Objetivos

General

Analizar y comparar la eficiencia del conjunto de casos de prueba generado al aplicar las propuestas recientes para pruebas funcionales desde la técnica “casos de prueba desde los casos de uso”.

Específicos

1. Describir en un marco ontológico unificado los objetivos, conceptos y métodos de los enfoques del estado del arte.
2. Definir los criterios para determinar la eficiencia del conjunto de casos de prueba para verificar los aspectos funcionales del software.

3. Aplicar las propuestas seleccionadas a un caso de estudio para verificar la eficiencia del conjunto de casos de prueba generado.
4. Comparar las propuestas, identificando su contribución a los objetivos, conceptos y métodos del estado del arte.

2. PROPUESTA METODOLÓGICA

1. Realizar una amplia búsqueda en la literatura acerca de las propuestas formuladas recientemente y orientadas a diseñar casos de prueba desde los casos de uso para las pruebas funcionales.
2. Definir el conjunto de conceptos –ontología– que permita describir de forma consistente los diferentes métodos.
3. Definir criterios de calidad para el conjunto de casos de prueba que permitan definir índices de comparación.
4. Hacer un análisis de cada propuesta para determinar claramente el proceso para diseñar casos de prueba, y poder clasificarlas en una matriz de datos.
5. Aplicar cada propuesta a un estudio de caso para determinar la eficacia del conjunto de casos de prueba generado.
6. Aprovechar las mejores prácticas encontradas en la aplicación de las propuestas para definir un método con el cual comprobar si es posible diseñar casos de prueba que garanticen que si son efectivos para unas pruebas lo serán para cualquiera, y documentarlo en una nueva propuesta.

En la Tabla 1.1 se detallan las actividades de esta metodología.

TABLA 1.1
Detalle de las actividades propuestas

Actividad	Resultado	Medio de verificación	Supuestos relevantes
Consultar bases de datos	Base de datos depurada de las publicaciones para análisis	Cumplimiento de las características de selección	Pocas o ninguna publicación Imposibilidad de consecución Fuentes poco fiables
Definir ontología	Matriz estructurada con las características de cada técnica analizada	Cumplimiento de las características de selección	Imposibilidad de verificación de características Poca información en la descripción de la propuesta
Diseñar y aplicar en estudio de caso	Matriz con la información resultante	Tabla de resultados	Errores de diseño Errores de aplicación Lecturas incorrectas
Analizar resultados	Matriz de comparación de resultados con datos cualitativos y cuantitativos	Tabla de resultados	Error en la selección de indicadores Falta de información relevante
Fusionar propuestas	Matriz de buenas prácticas	Nueva propuesta Documento de la propuesta	Errores de diseño Errores de aplicación Lecturas incorrectas
Informe final	Documento de tesis	Documento digital de acuerdo con el formato de la universidad	Tiempo insuficiente Funciones distractoras
Publicar artículos	Artículo para revista indexada	Contenido estructurado para cumplir las normas de la revista	No aceptación Tiempo de estudio extenso

3. ANTECEDENTES

En la industria y en los entornos académicos y de investigación actuales se acepta ampliamente el paradigma de la Programación Orientada por objetos –POO–, y es habitual expresar la funcionalidad de un sistema utilizando lineamientos como los diagramas UML para los casos de uso, por lo que dichos diagramas se convierten en una base adecuada para comenzar a estructurar

las pruebas del sistema. Por otro lado, existen sistemas considerados de software crítico, como los del área de la salud y los del transporte aéreo, cuyos requisitos deben especificarse mediante lenguajes formales, por lo que requieren de ingenieros con una alta preparación y sus costos de desarrollo son elevados. Es por esto que no se encuentran muchos trabajos que describan este tipo de especificación, además, porque para la mayoría de proyectos la ingeniería de software tradicional no utiliza requisitos altamente elaborados, y la mayoría de propuestas utilizan la especificación en casos de uso, descritos en lenguaje natural, como base para diseñar el conjunto de casos de prueba.

Este trabajo parte de una serie de estudios previos acerca de la temática para analizarlos y aplicarlos, y con los resultados estructurar una nueva propuesta para diseñar casos de prueba desde los casos de uso. En esta sección se describen los trabajos recientes relacionados con el tema, se describe su metodología, y se hacen comparaciones y análisis semejantes a lo que aquí se proponen.

En el trabajo de Ledru *et al*. (2001), la noción del propósito de la prueba se describe como la forma de especificar un conjunto de casos de prueba; además, describen cómo explotar los propósitos de la prueba utilizando varias herramientas para automatizar la generación de casos de prueba. Detallan las relaciones más importantes que permiten enlazar los propósitos de la prueba, los casos de prueba y los requisitos especificados; y exploran las similitudes y las diferencias entre la especificación de casos de prueba y la especificación de los programas. Este estudio es una recopilación interesante ya que describe procedimientos de cómo automatizar la aplicación de los casos de prueba, un área que no se aborda en esta tesis pero que se plantea como trabajo futuro.

El problema que aborda Vegas (2002) en su tesis se relaciona con una de las necesidades sentidas de la industria del software: “poder ayudar a los desarrolladores a mejorar la selección de técnicas de prueba” (Serna, 2010). En su tesis describe una comparación entre las técnicas más recientes, y finalmente elabora un repositorio detallado con la información resultante, pero en el que se centra en sus aspectos pragmáticos. Este estudio aporta una herramienta conceptual a los desarrolladores con la que pueden seleccionar una u otra técnica de prueba, pero no estructura un modelo de indicadores para valorar cuantitativamente cada una de las herramientas seleccionadas. Su principal aporte se aprecia en la pragmática, y aunque útil para los equipos de pruebas no presenta mayor interés para estructurar las pruebas desde las fases iniciales del ciclo de vida de los productos software.

Debido a que es necesario estructurar una forma efectiva para diseñar casos de prueba desde la perspectiva del cuerpo del conocimiento, Kaner (2003) concluye que la complejidad del diseño de casos de prueba se debe a que 1) los casos de prueba ayudan a descubrir información; 2) de acuerdo con el tipo de información es más eficaz seleccionar diferentes tipos de pruebas; y 3) si los casos de prueba son efectivos en una amplia variedad de formas, ¿cómo hacer para que lo sean en todas ellas? Además, reflexiona acerca de que los probadores tienden a crear casos de prueba de acuerdo con ciertos estilos de pruebas, como las pruebas de dominio o basadas en riesgos, aunque su concepción sea diferente. Este trabajo describe una serie de actividades documentadas para cubrir uno de los problemas de los casos de prueba: la falta de información suficiente para su diseño. Aunque es un aporte novedoso y del que se ha aprovecha el criterio que describe para la selección de datos, este trabajo no compara técnicas o propuestas, ya que su objetivo es indicar qué información necesita un buen caso de prueba y cómo obtener el mejor resultado de él.

El Instituto IESE de Alemania publicó un reporte técnico (Denger & Medina, 2003) en el que detalla el resultado de una búsqueda en la literatura cuyo objetivo fue identificar el estado actual de las técnicas para generar casos de prueba con base en los requisitos de usuario. El documento describe el proceso aplicado por los investigadores para realizar el estudio y presenta los resultados en términos de las referencias encontradas: 12 propuestas que van desde 1988 hasta 2002. Aunque contiene un resumen de los artículos más relevantes sobre el tema de estudio, aquellos que describen técnicas para derivar los casos de prueba desde los casos de uso descritos en lenguaje natural, no presenta estudios de caso prácticos o procesos de aplicación, ni realiza estudios de análisis comparativo. Además, muchas de las propuestas son antiguas y han pasado en la historia sin un riguroso estudio o seguimiento, y muchas otras, que para esta tesis son relevantes, no fueron tenidas en cuenta.

Burnstein publicó en 2003 un libro con una fuerte orientación a los procesos en ingeniería y a promover el crecimiento y el valor de las pruebas de software como una profesión, e introduce conceptos como gestión de pruebas y áreas técnicas, y describe cómo diseñar casos de prueba de acuerdo con esa concepción. Para esta investigación se referencia especialmente los capítulos 4 y 5, *Strategies and methods for test case design*, dada su orientación a procesos prácticos para diseñar y comparar casos de prueba. Un trabajo, que se tomó como guía para el logro de algunos de los objetivos de esta tesis, fue el de Gutiérrez *et al*. (2004), ya que describe las pruebas como una actividad paralela a las fases del ciclo de vida del software. Para ellos, las pruebas se utilizan generalmente para verificar la funcionalidad e integridad de los sistemas, y se aplican normalmente al finalizar el ciclo de vida del desarrollo, aunque sea posible planificarlas y aplicarlas desde las primeras etapas. Describen algunas propuestas para desarrollar el plan de pruebas del sistema, y hacen un análisis comparativo de ellas; también, analizan las fortalezas y debilidades de cada propuesta evaluada, con lo que deciden cuál propuesta adoptar para diseñar casos de prueba desde los requisitos iniciales. Este estudio no incluye una propuesta para estructurar un método con base en los resultados de los análisis.

La conclusión a la que llegan Juristo *et al*. (2004) es que para alcanzar resultados más estructurados de las pruebas, es necesario administrar adecuadamente el conocimiento en las disciplinas ingenieriles; pero que el conocimiento que se utiliza en la ingeniería del software tradicional presenta un nivel de madurez relativamente bajo, dado que es muy poco lo que se trabaja e investiga en áreas clave como las técnicas de prueba del software. Dado que estas técnicas se utilizan para diseñar los casos de prueba, que son el núcleo de la actividad misma, se requiere incrementar ese nivel de madurez para alcanzar desarrollos más fiables y eficaces. Este trabajo, mediante un análisis de los estudios empíricos existentes sobre las pruebas técnicas experimentales, analiza el nivel de madurez del conocimiento que presentan actualmente. No pasa de ser un reporte histórico que, aunque útil para esta investigación en cuanto a su amplia cobertura, no ofrece datos comparativos de la eficiencia o de la calidad del conjunto de casos de prueba que cada técnica genera.

Un trabajo cuyo proceso es parecido al que se aplica en esta tesis es el de Wagner (2004), quien compara la eficiencia de cuatro propuestas populares para detectar anomalías funcionales en el software, y describe los resultados obtenidos luego de aplicarlas en un estudio de caso. Utiliza métricas basadas en la cobertura de código, y un contador de errores; además, introduce la métrica de intensidad de la anomalía para evaluar la fiabilidad del software. Este proceso lo realiza sobre un sistema funcional de tres años de desarrollo, en el que busca errores con el objetivo de analizar la eficiencia de las técnicas seleccionadas. Aunque su utilidad en cuanto a indicadores y métricas es invaluable, este trabajo no estructura una nueva metodología para

aprovechar las fortalezas de las propuestas analizadas, algo que se explica dado el reducido número de elementos en la muestra.

En el estudio de Seo & Choi (2006) se presenta una comparación empírica de cinco técnicas de prueba de caja negra, y se detallan los diferentes resultados de aplicación luego de ponerlas en práctica en un laboratorio. Concluyen que las técnicas de prueba de caja negra verifican diferentes niveles del código, y que el probador puede considerar la combinación de los métodos de prueba para lograr resultados más eficientes, como la combinación de casos de uso extendidos con técnicas de pruebas en OCL. Aunque uno de sus aportes fundamentales es que las pruebas funcionales deben probar todas las especificaciones, y ser uno de los pocos trabajos que aplica las pruebas funcionales también al código, no detalla el proceso para seleccionar indicadores para el análisis, no tiene en cuenta otras propuestas, la muestra es pequeña, y no propone cómo aprovechar las mejores prácticas de las propuestas encontradas en el análisis.

Para determinar qué características debe tener una propuesta de pruebas que la hagan eficiente y de buena calidad, Gutiérrez *et al*. (2006) realizaron una investigación y presentaron los resultados en un artículo. La muestra del estudio fueron 13 propuestas para generar casos de prueba desde los requisitos funcionales, y parte de la idea de que el criterio de calidad más importante de un sistema software es el nivel en que satisface las necesidades de los usuarios. Concluyen que la mejor forma de verificar y mejorar el grado de cumplimiento de los requisitos es a través de la prueba de sistema, para lo que es muy eficaz derivar los casos de prueba desde los requisitos funcionales. El número de elementos de la muestra constituye uno de los puntos más fuertes de este trabajo, lo mismo que la amplia lista de indicadores que aplica. Pero no indica cómo construyeron la lista de indicadores ni la métrica aplicada, no describe el estudio de caso que utilizaron, y no describen una propuesta en la que aprovechen las fortalezas encontradas en las propuestas.

La mayoría de libros consultados en el estado del arte y el marco teórico de esta investigación se orientan a explicar *qué* hacer en pruebas, pero no describen *cómo* hacerlo. Una excepción a esto es el texto de E. Perry (2006), cuya orientación es a describir, desde diversos enfoques, *cómo* estructurar y aplicar los casos de prueba en los procesos de verificación del software. Además, proporciona procedimientos, plantillas, listas de comprobación y cuestionarios de evaluación, necesarios para llevar a cabo una eficaz y eficiente prueba del software. De su contenido se toma la tercera parte, *The Seven-Step Testing Process*, en la que describe un proceso de prueba de software con base en la experiencia acumulada luego de trabajar con más de 1000 organizaciones afiliadas al Instituto de Garantía de Calidad de los EEUU. Otra cuestión importante en las pruebas es poder gestionar eficientemente el diseño de los casos de prueba, temática que Kamde *et al*. (2006) desarrollan en su trabajo. Dicho artículo describe cómo evitar la pérdida de gestión por diseñar casos de prueba pobres o deficientes; ofrece una serie de recomendaciones prácticas acerca de cómo mejorar la productividad, la facilidad de uso, la fiabilidad y la gestión de ese proceso de diseño. Una vez que explica qué son y el porqué de los casos de prueba, utiliza una lista de verificación para identificar las áreas de riesgo, con la que se mejoran los casos de prueba diseñados y se mejoran los procesos para diseños posteriores. Incluye un estudio de caso en el que describe cómo utilizar los casos de prueba para mejorar la capacidad de prueba y su productividad, la forma de resolver los problemas comunes al probar la calidad de los mismos, y cómo proteger los caso de prueba que se llevarán a la práctica.

El diseño de escenarios eficientes y bien documentados, una dificultad a momento de estructurar completamente el conjunto de casos de prueba, es el área que desarrollan Gutiérrez *et al*. (2007).

Este estudio describe una serie de procesos para generar objetivos de prueba, escenarios de prueba y variables funcionales, ideas fundamentales para estructurar una propuesta de prueba. Aunque expresan los casos de uso de forma narrativa, contrario a formalizarlo lo más posible, automatizan el proceso con dos herramientas de apoyo, y justifican la aplicación de las pruebas desde las fases iniciales del ciclo de vida. Resaltan la falta de enfoques para obtener objetivos de prueba y casos de prueba desde casos de uso documentados formalmente.

Luego de analizar estos trabajos previos, la conclusión es que existen pocos estudios que describan comparativamente procesos para generar casos de prueba desde los casos de uso para las pruebas funcionales, mientras que existe suficiente soporte para aplicar cada técnica de prueba individualmente. Todo esto sustenta el desarrollo de un estudio acerca del estado actual en esta área de investigación, lo que se convierte en el marco de referencia para la elaboración de la presente tesis.

REFERENCIAS

- Allott, S. (2007). "Testing Techniques: Are they of any Practical Use?" *Academic and Industrial Conference Practice and Research Techniques, MUTATION*. Cumberland Lodge, UK, pp. 35-39.
- Basili, V. R. & Selby R. W. (1987). "Comparing the Effectiveness of Software Testing Strategies". *IEEE Transactions on Software Engineering*, Vol. 13, No. 12, pp. 1278-1296.
- Baudry, B., Fleurey F., Jezequel J-M. & Traon Y. L. (2002). "Automatic test cases optimization using a bacteriological adaptation model: Application to .NET components". *17th IEEE International Conference on Automated Software Engineering, ASE'02*. Edinburgh, UK, pp. 253-257.
- Beizer, B. (1990). "Software Testing Techniques". New York: Van Nostrand Reinhold.
- Beizer, B. (1995). "Black-Box Testing: Techniques for Functional Testing of Software and Systems". USA: Wiley.
- Binder, R. V. (2000). "Testing object-oriented systems: models, patterns and tools". New York: Addison-Wesley.
- Boehm, B. W. (1973). "The high cost of software". *Symposium on High Cost of Software*. Monterey, California, pp. 27-40.
- Brown, J. R. & Lipow M. (1973). "The quantitative measurement of software safety and reliability". *TRW Report SDP-1176*. TRW Software Series.
- Burnstein, I. (2003). "Practical software testing. A process-oriented approach". New York: Springer-Verlag.
- Denger, C. & Medina M. M. (2003). "Test case derived from requirement specification". *Fraunhofer IESE. IESE-Report 033.03/E*. Kaiserslautern, Germany.
- Goodenough, J. B. (1979). "A survey of program testing issues". In P. Wegner (Ed.) *Research Directions in Software Technology*. Massachusetts: MIT Press, pp. 316-340.
- Gutiérrez, R. J. J., Escalona C. M. J., Mejías R. M. & Torres J. V. (2006). "Generation of test cases from functional requirements: a survey". *Workshop on System Testing and Validation*, Vol. 4, No. 4, pp. 117-126.
- Gutiérrez, R. J. J., Escalona C. M. J., Mejías R. M. & Torres J. V. (2004). "Comparative analysis of methodological proposes to systematic generation of system test cases from system requisites". *Workshop on Systems Testing and Validation*. Paris, France, pp. 151-160.
- Gutiérrez, R. J. J., Escalona C. M. J., Mejías R. M. & Torres J. V. (2006). "Generation of test cases from functional requirements: a survey". *Workshop on System Testing and Validation*, Vol. 4, No. 4, pp. 117-126.
- Gutiérrez, R. J. J., Escalona C. M. J., Mejías R. M. & Torres Z. A. H. (2007). "Using use case scenarios and operational variables for generating test objectives". *5th Workshop on Systems Testing and Validation*. Paris, France, pp. 23-32.
- Hetzel, W. C. (1988). "The Complete Guide to Software Testing". Massachusetts: Weley.
- Horowitz, E. (1975). "Practical Strategies for Developing Large Software Systems". Massachusetts: Addison-Wesley.
- IEEE (1983). "IEEE Standard for Software Test Documentation". Std 829-1983.
- IEEE (1990). "IEEE Standard Glossary of Software Engineering Terminology". Std. 610.12-1990.
- Juristo, N., Moreno A. M. & Vegas S. (2004). "Reviewing 25 Years of Testing Technique Experiments". *Empirical Software Engineering*, Vol. 9, No. 1-2, pp. 7-44.
- Kamde, P. M., Nandavadekar V. D. & Pawar R.G. (2006). "Value of test cases in software testing". *Management of Innovation and Technology*, Vol. 2, pp. 668-672.
- Kaner, C. (2003). "What is a good test case?" *StarEast conference 03*. Orlando, USA, pp. 34-50.
- Kaner, C., Falk J. & Nguyen H. Q. (1999). "Testing computer software". New York: John Wiley & Sons.
- Kobrosly, W. & Vassiliadiset V. (1988). "A Survey of Software Functional testing Techniques". *IEEE S.T.T. Conference*. Binghamton, New York, USA, pp. 127-134.
- Ledru, Y., du Bousquet L., Bontron P., Maury O., Oriat C. & Potet M. L. (2001). "Test purposes: adapting the notion of specification to testing". *16th IEEE International Conference on Automated Software Engineering, ASE'01*. San Diego, USA, pp. 127-134.
- Lewis, W. E. (2000). "Software testing and continuous quality improvement". USA: CRC Press LLC.
- Myers, G. J. (2004). "The Art of Software Testing". New York: John Wiley & Sons.
- Patton, R. (2000). "Software testing". Indianapolis: Sams Publishing.
- Perry, W. E. (2006). "Effective methods for software testing". USA: Wiley Publishing Inc.
- Radatz, J. W. (1981). "Analysis of V&V data". *Rome Air Development Center Report RADC-TR-81-145*. Logicon Inc. San Pedro, California.
- Seo, K. I. & Choi E. M. (2006). "Comparison of five black-box testing methods for Object-Oriented software". *Fourth International Conference on Software Engineering Research. Management and Applications*. Seattle, USA, pp. 213-220.
- Serna, M. E. (2010). "Métodos formales en la industria y la investigación". En prensa.

- Vegas, H. S. (2002). “Esquema de caracterización para la selección de técnicas de pruebas de software”. *Tesis doctoral*. Universidad Politécnica de Madrid.
- Wagner, S. (2004). “Efficiency Analysis of Defect-Detection Techniques”. *Technical Report TUMI-0413*. Institut für Informatik, Technische Universität München.
- Wolverton, R. W. (1974). “The cost of developing large scale software”. *IEEE Transactions on Computers*, Vol. 23, No. 6, pp. 615-636.

II. MARCO TEÓRICO

1. ANOMALÍAS DEL SOFTWARE

Aceptar la existencia y necesidad del software y la importancia que ha alcanzado en el desarrollo de la Sociedad del Conocimiento actual no es está en discusión. Hoy existen computadores y software para todo tipo de actividades en las empresas y para las personas, quienes en su mayoría tienen dependencia constante del internet.

Construir software es un proceso complejo y para lograr que el producto final satisfaga los objetivos para los que se construye y que se pueda considerar terminado, es necesario aplicar procesos establecidos y difundidos en las continuas investigaciones de las Ciencias Computacionales, especialmente acerca de los temas que tienen relación con las metodologías de desarrollo. Este proceso se conoce comúnmente como ciclo de vida del producto y está constituido por una serie de fases que debe "vivir" el producto hasta su entrega al cliente, y con las cuales es posible estructurar y presentar una solución a las necesidades que previamente el cliente manifiesta.

Por lo tanto, el software es un producto del intelecto humano que, como todos los procesos semejantes, no está libre de errores y mal funcionamiento. Esas fases del ciclo de vida, tradicionalmente denominadas requisitos, análisis, diseño, desarrollo e implementación, son el resultado del trabajo entre humanos; es una labor entre equipos de personas que interactúan, comparten, discuten y proponen alcances a las soluciones que ofrecerán a los problemas computacionales; es un diálogo contante entre el cliente, que manifiesta sus necesidades, y los ingenieros, que intentan comprenderlas y ofrecer soluciones.

Los requisitos del software expresan las necesidades del cliente y pueden ser funcionales o no funcionales. Los funcionales determinan los resultados generados por las diferentes operaciones del software como consecuencia de la transformación de los datos o valores de entrada, y recibidos antes de ejecutar dichas operaciones. Los no funcionales –requisitos de calidad– especifican la forma como se comporta el software durante el proceso de transformación de los datos en resultados, incluyendo, entre otros, tiempos de respuesta, capacidad de carga, distribución en el tiempo y el espacio de los lugares de ingreso y recepción de datos, facilidad de uso y seguridad de la información.

Dichas necesidades del cliente, en un proceso riguroso de desarrollo de software, se especifican en un documento que debe contener la información necesaria de tal forma que sea posible definir lo que el producto será, lo que hará y cómo se verá. Aunque no se encuentra una estructura generalizada que indique cómo presentarlo, es un documento técnico que establece de forma clara y precisa todas las características, materiales y servicios, necesarios para desarrollar el producto que requiere el cliente (Patton, 2000). La importancia de este documento radica en que es el origen de muchos de los problemas que el software presenta. En la fase de análisis, por ejemplo, se genera la especificación de requisitos, que incorpora el conjunto de necesidades del cliente –análisis del problema–, y la manera como el software debe comportarse para satisfacerlas –requisitos del software.

La base de este proceso es la información, que se elicita en las primeras fases –ingeniería de requisitos–, y posteriormente se estudia, condensa e interpreta –se especifica–, para decidir exactamente qué características deberá tener el producto terminado (Nebut *et al*., 2004). Cuando

existe una discrepancia entre el valor especificado de entrada al sistema y el valor especificado de salida, es cuando se presenta el mal funcionamiento del software. Puede ser desde un simple inconveniente, como cuando un videojuego no funciona adecuadamente, hasta algo catastrófico, como cuando el resultado es la pérdida de vidas (Leszaca *et al*., 2002). El error humano es inevitable, incluso cuando usuarios experimentados realizan tareas sencillas (Hollnagel, 1993), (Lazonder & van Der Meij, 1995), (Virvou, 1999), por lo que las anomalías no pueden tomarse a la ligera. Debido a esto, la baja calidad del software, en la que los errores son un hecho, se ha convertido en un problema serio que afecta a desarrolladores y usuarios. En promedio un programa contiene entre diez y veinte errores por cada mil líneas de código (Levendel, 1990), y los grandes sistemas pueden contener miles e incluso decenas de miles de errores potenciales en su primera entrega (Patton, 2000).

El software funciona mal tanto si no lleva a cabo las tareas requeridas como si las lleva a cabo de forma incorrecta. Por definición, un error de software es una imperfección estructural en un programa que puede conducir a la eventual falla del sistema (Perry, 2006). Es una característica física del sistema en la que su tipo y extensión pueden ser medidos de la misma forma que se miden las propiedades de los sistemas físicos tradicionales. Una anomalía en el software se presenta cuando una o más de las siguientes condiciones es cierta (Patton, 2000):

1. No hace lo que la especificación dice que debe hacer
2. Hace lo que la especificación dice que no debe hacer
3. Hace algo que en la especificación no se menciona
4. No hace algo que la especificación no menciona, pero que debería hacer
5. Es difícil de entender, difícil de usar o es lento.

Las condiciones 1 y 2 se presentan debido a que los requisitos expresados en el análisis no se trasladan adecuadamente a las fases siguientes del ciclo de vida, lo que implica que el software codificado o el hardware no responda de acuerdo con los mismos. Para este trabajo, los problemas que se generan en esta situación se consideran problemas de Verificación. Las condiciones 3 y 4 se presentan porque la especificación puede ser incompleta, ambigua o contradictoria, y aquí se consideran problemas de Validación. La condición 5 esta embebida en las anteriores, ya que hace referencia a anomalías frente a requisitos no funcionales.

Para revisar que las metas y objetivos del software se están logrando a través del ciclo de vida, se ha propuesto un proceso conocido como de Verificación y Validación –*V&V process*. La Verificación es un proceso de control para asegurar que el software cumple con su especificación. La Validación debe distinguirse de la Verificación, ya que es un proceso para comprobar que el software satisface las necesidades del usuario Para diferenciar ambos procesos, Boehm (1979) propuso la siguiente definición: Validación debe responder a la pregunta ¿*Estamos construyendo el producto correcto*? y Verificación a ¿*Estamos construyendo correctamente el producto*? Es decir, validar es controlar que el producto satisface los requisitos del cliente, mientras que verificar es controlar que el producto cumple su especificación inicial.

La Verificación es cualquier actividad dirigida a la evaluación de un atributo o capacidad de un programa o sistema, y la posibilidad de determinar que cumple con lo especificado (Hentzel, 1988). Es un proceso que asegura que el producto software se desarrolla de manera correcta; el software debe confirmar las especificaciones predefinidas y, dado que el desarrollo del producto pasa por diferentes etapas, la verificación se hace para asegurar que todas las especificaciones requeridas se cumplen.

Para esta investigación, una anomalía es la manifestación de un error; pero en el contexto no aporta una distinción mayor. Así, la clasificación de errores puede asimilarse a la de anomalías. Aunque la industria del software reconoce esta cuestión, existen pocas investigaciones acerca de qué son y cómo se clasifican (Kay, 2007), pero lo cierto es que casi siempre existen de forma moderada en cualquier sistema. Otro aspecto es que no es posible realizar una búsqueda completa de anomalías en un sistema, debido a que todos los posibles valores especificados de entrada se deben probar, y la tarea se convierte en imposible (Weiss & Basili, 1985).

1.1 Taxonomía de las anomalías

Existen varias propuestas de taxonomías para las anomalías, como las de Weiss (1979), Glass (1981), Basili & Perricone (1984), Ostrand & Weyuker (1984), Basili & Selby (1987), Richardson & Thompson (1988) y Marick (1990); inclusive, algunos de ellos intentaron determinar las técnicas más eficientes para mostrar las anomalías. Knuth (1989), Beizer (1990) y el estándar IEEE 1044-1993 (1994) clasifican las anomalías para sistemas no Orientados por Objetos, y las identifican de acuerdo con el tipo de prueba que se planifica para el programa. En este paradigma también se ha presentado algunas clasificaciones: Purchase & Winder (1991), Overbeck (1993), Firesmith (1993), Huffman (1994), pero no es posible hallar una propuesta de clasificación universal. Beizer (1990) propone: funcionales, estructurales, de datos y de las pruebas; y Lewis (2009) propone: de requisitos, de características, de funcionalidad, de estructura, de datos, de implementación, de codificación, de integración, de arquitectura de software, del sistema, y de la prueba. Pero, más importante que adoptar la taxonomía correcta es adoptar alguna taxonomía, y utilizarla como marco estadístico sobre el que se basa la estrategia de las pruebas.

Una anomalía cualquiera se puede ubicar en una u otra taxonomía en función de su historia y las decisiones de los desarrolladores o probadores, y esa dificultad es lo que ha generado la diversidad de taxonomías. Debido a esa diversidad de propuestas y a la dificultad para clasificarlas adecuadamente, en este trabajo se propone una clasificación simplificada para las anomalías según el tipo de especificación: funcionales, no funcionales, de análisis de diseño, de codificación y de pruebas.

1.2 Causas de las anomalías

Las causas de las anomalías son diversas ya que dependen del tipo mismo de la anomalía. Igualmente, pueden aparecer e introducirse al sistema en el desarrollo de cualquier fase del ciclo de vida, o por acciones que ejecutan los actores involucrados en el proceso.

- Las *anomalías funcionales* se originan en una discrepancia de la concreción de un requisito de una fase anterior a una fase siguiente, por lo que su modelo no expresa lo especificado en la anterior, y reflejará un error o equivocación en la fase posterior.
- Las *anomalías no funcionales* tienen diversos orígenes: deficiencias en el hardware, en las comunicaciones o en otro software con el que el programa comparte procesos.
- Las anomalías de *análisis de diseño* son causadas por una interpretación inadecuada de las necesidades del cliente o porque el documento de especificación no es claro; por lo que el diseñador estructura una solución no coherente con lo especificado.
- Los mismos desarrolladores insertan errores en los sistemas, ya sea de forma consciente o por omisión. Los errores consientes son líneas implementadas en el código que no hacen parte de la especificación o diseño, y los errores por omisión son especificaciones que no se implementan, por olvido o desconocimiento.
- De forma general se introduce un nuevo error por cada tres corregidos (Levendel, 1990).

La importancia de una anomalía depende de su frecuencia, costo de corrección, costo de instalación y sus consecuencias.

- *Frecuencia*. ¿Con qué frecuencia ocurre ese tipo de anomalía?
- *Costo de corrección*. ¿Cuál es el costo de corregir el error que origina la anomalía después de encontrado? Este costo es la suma de dos factores: el costo de descubrimiento y el costo de la corrección. Los costos se incrementan dramáticamente si los errores se descubren en fases avanzadas del ciclo de vida, y depende del tamaño del sistema, ya que cuanto mayor sea más costará corregir el error.
- *Costo de instalación*. Depende del número de instalaciones: pequeño para un programa de usuario único, pero alto para un sistema que opera en red. El costo de instalación puede dominar los demás costos, ya que corregir un error simple y distribuir la corrección podría superar el costo del desarrollo de todo el sistema.
- *Consecuencias*. ¿Cuáles son las consecuencias de las anomalías? Es posible medirlas por el promedio de las que reportan los usuarios. Las consecuencias se pueden clasificar en leves, moderadas, molestas, disturbios, graves, muy graves, extremas, intolerables, catastróficas, infecciosas (Whittaker, 2000).

La frecuencia no depende de la aplicación o el entorno, pero la corrección, la instalación y las consecuencias sí, por lo que los probadores y aseguradores de calidad se deben interesar por la importancia, no por la frecuencia (Levendel, 1990).

2. VERIFICACIÓN DEL SOFTWARE

El costo de corrección de los errores es logarítmico, es decir, se incrementa en diez veces a medida que avanza en las fases del ciclo de vida del producto. Según Patton (2000), la corrección de un error encontrado en las primeras fases, cuando se escribe la especificación, podría costar 10 centavos de dólar; si se detecta luego de la codificación costaría entre 1 y 10 dólares; y si lo detecta el cliente, podría sobrepasar los 100 dólares.

En lugar de probar el software para buscar errores detectando las anomalías que producen lo mejor es revisarlo en todo el ciclo de vida. Por lo que el proceso de Verificación es una actividad que debe realizarse paralelamente al desarrollo del ciclo de vida del software, ya que llevarla a cabo al final, cuando el producto esté "terminado", puede ser un proceso que, por falta de tiempo, se realice sin las condiciones de calidad necesarias e incrementa el costo de desarrollo del producto. La detección constante de errores que originan anomalías es una fuente de frustración y temor, y las empresas de software suelen gastar hasta el 80% de su presupuesto de desarrollo en asegurar el control de calidad (Pfleeger, 2001). Sin embargo, a pesar de que los desarrolladores invierten innumerables horas tratando de evitarlos, la complejidad del desarrollo del software ocasiona que, mucho después que se entrega a los usuarios, el producto siga mostrando errores, como lo demuestra la continua aparición de actualizaciones luego de su lanzamiento inicial.

Un aspecto importante es que no es posible garantizar o probar que un sistema no tiene errores que originen anomalías, sólo es posible mostrar que los tiene; es decir, no encontrar errores en un producto software no prueba que no los tenga. Esta premisa sustenta la propuesta de realizar las pruebas de forma paralela al ciclo de vida, ya que las anomalías que se detectan al final del ciclo de vida obligan regresar a fases previas para solucionarlas, por lo que *evitar errores* es más costoso que *remover errores* (Weitzenfeld, 2005).

2.1 Historia de la verificación del software

La historia de la Verificación del software comienza con el diseño del primer lenguaje de programación moderno, el Fortran, y el primer compilador en abril de 1957; y desde entonces se ha desarrollado de forma paralela a la historia del hardware y del software. A continuación se describe la evolución de esta actividad de acuerdo con Lewis (2009), y se establece un punto de vista acerca de la dirección de las investigaciones en el área.

En la década de los años 50 se definió Verificación como lo que hacen los desarrolladores para encontrar errores en sus programas; cerca de los 60 se pensó en realizar pruebas totales a los sistemas, pero se concluyó que la prueba "exhaustiva" era teórica y prácticamente imposible. En poco tiempo se concluyó que era muy difícil aplicar este tipo de prueba ya que: 1) el dominio de las entradas del programa era muy amplio, 2) existían muchos posibles caminos de entrada, y 3) el diseño y la especificación eran difíciles de probar.

Al iniciar los 70, el desarrollo de software maduró y evolucionó en gran medida; se acuñó el término *Ciencia Computacional* para referenciar todo los procesos científicos relacionados con la computación; y la prueba de software pasó a definirse como lo que se hace para *demostrar la correctitud de un programa*. Por esta misma época se propuso verificar al software mediante una *prueba de correctitud*, pero se concluyó que era una técnica ineficiente de prueba ya que, en la práctica, demostrar que el programa es correcto era demasiado lento e insuficiente. A finales de la década la percepción de prueba pasó a ser *un proceso de ejecución de un programa con la intención de encontrar un error, sin demostrar que funciona*, y se aceptó que una prueba es correcta si tiene una alta probabilidad de encontrar un error aún sin descubrir, y que es exitosa cuando descubre ese tipo de error.

En la década de 1980, se planteó la necesidad de una metodología de prueba que incluyera comentarios a lo largo del ciclo de vida, y que debería ser un proceso planificado; además, surgieron herramientas para automatizar las pruebas, y para mejorar la eficiencia y la calidad de las aplicaciones. A principios de 1990 las herramientas para administrar pruebas permitieron gestionarlas desde los requisitos y el diseño hasta la codificación y la entrega; se reconocieron las ventajas de diseñar y aplicar las pruebas desde las fases iniciales del desarrollo, y de diseñarlas y aplicarlas como un proceso paralelo desde las fases iniciales del desarrollo. El concepto de prueba se convirtió entonces en *la planificación, el diseño, la construcción, el mantenimiento y la ejecución de las pruebas en entornos precisos*.

En la década de 2000, se reforzó el concepto de desarrollo paralelo y se fortaleció el concepto de prueba como un proceso paralelo al ciclo de vida del desarrollo. Con el establecimiento del Desarrollo Orientado por Objetos surgió la tendencia de aprovechar la información que contiene el documento de Casos de Uso, y se desarrolló la técnica de pruebas conocida como *Casos de Prueba desde los Casos de Uso* (Copeland, 2004). Esta técnica aprovecha los caminos de ejecución y los escenarios de prueba, y aplica las mejores prácticas y la administración eficiente para diseñar pruebas.

Por esta misma época se reforzó la definición de dos conceptos clave en la prueba del software: 1) *camino de ejecución*, que es la ruta que puede tomar un dato para recorrer un programa; es decir que colocando un valor a la entrada del programa es posible conocer el recorrido que tendrá hasta llegar a la salida. Para representar los caminos se utilizan grafos dirigidos, construidos a partir de la especificación de requisitos en los Casos de Uso (Whittaker, 2000). Dos caminos se consideran diferentes si en un momento dado toman diferentes direcciones, ya sea por un error

del código o por una alteración del hardware. Myers (2004) demostró que, incluso un programa *simple* puede tener un número de caminos bastante amplio. 2) *escenario de prueba*, que es la combinación de valores de entrada, grafos de ejecución, especificaciones, salidas esperadas, y las secuencias en que serán ejecutados. Son argumentos de prueba que aseguran que los caminos en el grafo de ejecución se recorren de principio a fin. Su base de diseño es la revisión de los requisitos funcionales y la selección de valores de entrada, para representar tanto situaciones normales como inusuales que pueden ocurrir en el software (Hartmann *et al.*, 2000).

La Tabla 2.1 contiene las características de la historia de la Verificación del software.

TABLA 2.1
Características históricas de la Verificación del software (Juristo *et al.*, 2004)

Tiempo/Concepto	Funcional	Estructural	Funcional/Estructural
1950: Pruebas para conocer si un programa satisface los requisitos **1957**: Diferencia entre depuración y pruebas **1975**: Teoría fundamental de selección de datos de prueba **1979**: Asociar pruebas con detección de errores **1983**: Pruebas para prevenir errores en verificación, validación y pruebas **1991**: Integrar las actividades de las pruebas al ciclo de vida del software	**1980**: Pruebas funcionales y abstracción de diseño **1990**: Pruebas basadas en lógica **1997**: Prueba funcional probabilística **2000**: Utilización de la documentación de Casos de Uso para generar casos de prueba	**1975**: Decantar el concepto de la selección de la prueba; Intentar demostrar que funciona **1976**: Camino a cómo seleccionar la prueba **1980**: Surge el concepto del domino de la prueba **1985**: Estrategia orientada al flujo de datos **1994**: Modelo basado en la cobertura para estimar fiabilidad **1997**: Prueba de integración basada en la descripción de la arquitectura	**1989**: Integrar pruebas basadas en la especificación y la implementación usando métodos formales **2000**: Prueba de integración basada en UML **2001**: Técnica integrada para software basado en componentes

2.2 Técnicas de Verificación

Luego de hacer el análisis a la historia de la Verificación se concluye que existen dos categorías de técnicas: La Verificación *dinámica,* que requiere ejercitar el software, tiene como objetivo revisar la sintaxis, las salidas, las interfaces, los cambios de estado o la lógica; y la Verificación *estática,* que no requiere ejecutar el sistema, tiene como objetivo analizar los requisitos, la especificación, el diseño o el código (Patton, 2000). La Verificación estática no hace parte del contenido de esta investigación.

Para la Verificación dinámica se han propuesto algunas técnicas de aplicación, pero en esta tesis se proponen las siguientes:

- El concepto de *métodos formales* involucra una serie de técnicas lógicas y matemáticas con las que es posible especificar, diseñar, implementar y verificar los sistemas de información (Serna, 2010). La palabra formal se deriva de la Lógica Formal, ciencia que estudia el razonamiento desde el análisis formal –de acuerdo con su validez o no validez–, y omite el contenido empírico del razonamiento para considerar sólo la forma –estructura sin materia. Los métodos formales más rigurosos aplican estas técnicas para comprobar los argumentos que se utilizan para justificar los requisitos u otros aspectos de diseño e implementación de un sistema complejo.

- En la *revisión por pares*, el software es ejercitado por una serie de personas con el objetivo de revisarlo, criticarlo y mejorarlo. Esta revisión tiene su correspondiente en la ley: *dados suficientes ojos, todo error es superficial*; lo que se interpreta como que: *con suficientes revisores, cualquier problema se puede resolver fácilmente*. "La aplicación de la revisión por pares tiene altos beneficios en el desarrollo de software, dado que permite encontrar anomalías mucho más rápido que por pruebas o por reportes de errores de los usuarios, minimizando tiempo, esfuerzo y costos asociados" (Raymond, 2001).

- La *animación del software* consiste en ejercitarlo bajo unas condiciones específicas controladas, que en conjunto se conocen como *pruebas*. La prueba del software es un proceso que requiere tiempo y presupuesto, por lo que su diseño, como ya se dijo antes, se debe planificar y estructurar con tiempo y paralelamente al desarrollo del ciclo de vida del sistema. "Las pruebas de software consisten en verificar dinámicamente la conducta del programa bajo un conjunto finito de casos de prueba, para luego comparar los resultados con lo que se esperaba" (Jorgensen, 2004). Existe diferentes tipos de animación para verificar el software: capacidad de carga, velocidad de respuesta, resultados ofrecidos, congestionamiento, entre otras; y existen técnicas de animación para detectar anomalías funcionales y otras para detectar anomalías no funcionales.

En esta tesis se trabaja alrededor de la técnica de Verificación mediante prueba del software, y se asume como la actividad que ejecutan los probadores para demostrar que el producto cumple con lo especificado.

3. LA PRUEBA DEL SOFTWARE

La prueba del software es un importante medio para evaluar el software con el objetivo de determinar su calidad, y es una práctica realmente eficaz para asegurar esa calidad en un sistema de software de complejidad no trivial. Además, es un área de investigación importante dentro de las Ciencias Computacionales y probablemente será cada vez más importante en el futuro. El objetivo general de la prueba es afirmar sistemáticamente la calidad del software mediante su animación bajo circunstancias cuidadosamente controladas.

Para realizar esta animación es necesario estructurar y diseñar un conjunto de casos de prueba, y que de acuerdo con Moore (1994), es:

1. Un *conjunto de valores de entrada*: recibidos de una fuente externa al código que se está probando.
2. *Condiciones de ejecución*: requeridas para ejecutar la prueba, por ejemplo, un cierto estado de la base de datos o la configuración de un dispositivo de hardware.
3. *Salidas esperadas*: resultados específicos que debe producir el sistema bajo prueba, y que deben estar especificados.

Una implicación importante para un caso de prueba es que debe ser razonablemente capaz de revelar errores. También se argumenta que una prueba necesariamente no se diseña para exponer errores, que su objetivo debe ser producir información que a menudo implica errores aunque no siempre, y que para evaluar el valor de un caso de prueba es necesario preguntarse qué tan bien proporciona la información que se busca (Sinha & Suri, 1999).

Debido a la importancia de la prueba del software y a que es un error ejecutarla como una fase al final del desarrollo del producto, se deben diseñar casos de prueba para cada una de las fase del

ciclo de vida, es decir, debería tenerse casos de prueba para cada especificación en cada fase del ciclo, desde los requisitos, el análisis, el diseño, hasta el desarrollo y la implementación. Esto se sustenta en el hecho de que el número de casos de prueba es incremental: mientras que probar la especificación inicial requiere algunos casos de prueba, la fase siguiente le suma una cantidad mayor y así sucesivamente, hasta llegar a la fase de implementación, cuando es el usuario quien detecta las anomalías del producto.

De acuerdo con el tipo de requisito que se desee probar, existen casos de prueba para requisitos no funcionales y para requisitos funcionales. En este trabajo se describe el proceso para aplicar una metodología que permita estructurar un conjunto de casos de prueba para los requisitos funcionales. El proceso consiste en: 1) seleccionar una característica del sistema, 2) seleccionar un conjunto de entradas de las que ejecuta la característica, 3) documentar las salidas esperadas y, 3) en lo posible, diseñar pruebas automatizadas para probar que las salidas encontradas y las esperadas son iguales. Se recomienda comenzar con las pruebas de más alto nivel –desde los requisitos–, y, progresivamente, alcanzar pruebas con más detalle (Lewis, 2009).

Existen diversas propuestas para clasificar las pruebas del software. De acuerdo con Lewis (2009), a continuación se describe la que generalmente acepta la comunidad:

- *Funcionales* o de Caja Negra. La presente investigación se centra en este tipo de prueba, en la que lo importante es probar la funcionalidad del programa contra la especificación funcional elicitada en los requisitos; para esto es necesario observar al sistema como una Caja Negra y completamente ajeno a su estructura interna (Agarwal *et al*., 2010). "Es un conjunto de entradas, condiciones de ejecución y salidas esperadas con un objetivo particular; la idea es ejercitar la ruta de un programa específico o verificar el cumplimiento de un requisito específico" (IEEE, 1990). Este tipo de prueba tiene la limitante de que no es factible probar todos los posibles casos de entrada, además, dado que no se conoce la estructura interna o la lógica del programa, podrían existir errores en caminos internos de ejecución, que no se ejecutan en las pruebas definidas e insertados deliberadamente por el desarrollador, que posiblemente no detectarán. El proceso para ejecutar este tipo de prueba es:

  1. Analizar los requisitos y sus especificaciones
  2. Seleccionar entradas válidas y no válidas de acuerdo con la especificación
  3. Determinar las salidas esperadas para cada entrada
  4. Diseñar los casos de prueba con las entradas seleccionadas
  5. Ejecutar los casos de prueba
  6. Comparar las salidas encontradas con las salidas esperadas
  7. Determinar si el funcionamiento del software en prueba es apropiado.

  Los casos de prueba en estas pruebas buscan demostrar que:
  1. Las funciones del software son operativas
  2. La entrada se acepta de forma correcta
  3. Se produce una salida correcta
  4. La integridad de la información externa se mantiene.

- *Estructurales* o de Caja Blanca. Las condiciones de esta prueba se diseñan para examinar los caminos de la lógica, con lo que es posible analizar la estructura interna del sistema. Una de sus ventajas es que son exhaustivas y que se centran en el código, por lo que se tienen mayor probabilidad de detectar los errores en todos los caminos de ejecución. Su desventaja es que

no comprueba la correctitud de las especificaciones, ya que se centra sólo en la lógica interna. Otro inconveniente es que no existe forma de detectar los caminos perdidos que, sin estar en el código, pueden recorrer los datos; y que tampoco es posible recorrer todas las posibles rutas lógicas a través de un programa, ya que implicaría un gran número de pruebas (Agarwal *et al*., 2010).

- *Funcionales/Estructurales* o de Caja Gris. El estudio de la especificación de requisitos por el probador y la comunicación con el desarrollador son necesarios para comprender la estructura interna del sistema, y clarificar las especificaciones ambiguas (Pfleeger, 2001). Este tipo de pruebas es una combinación de las dos anteriores.

3.1 Pruebas funcionales del software

Son estrategias, metodologías o políticas que se proponen para aplicar los procesos de prueba a un programa o sistema y, como todas las pruebas, consumen al menos la mitad de la mano de obra utilizada para desarrollar el sistema (Boehm, 1973), (Brown, 1973), (Wolverton, 1974), (Goodenough, 1979). "Estas pruebas sólo pueden demostrar la presencia de errores, no su ausencia" (Dijkstra *et al*., 1972).

El desarrollo de pruebas funcionales tiene dos objetivos: 1) mostrar, al desarrollador y al cliente, que el software cumple con la especificación, y 2) descubrir las anomalías en el sistema que generan comportamiento incorrecto, no deseado o que incumplen la especificación. Su función es describir metodologías que permitan aplicar las técnicas o métodos necesarios para realizar una prueba determinada; previamente se debe acordar, por parte de los probadores, si se hará una prueba estática o dinámica al sistema.

3.2 Niveles de las pruebas funcionales

Se refieren a la cobertura de la prueba planificada que se desea aplicar en el software y, conceptualmente, se pueden distinguir tres niveles: de unidad, de integración y de sistema. No se conoce un modelo que determine cómo aplicar un nivel determinado, ni cómo asumir que alguno de ellos tenga mayor importancia (Beizer, 1990), (Pfleeger, 2001), (Jorgensen, 2004).

- Pruebas *de unidad o de módulo*. Es un proceso de prueba individual a los subprogramas, subrutinas o procedimientos de un sistema. En lugar de una prueba completa al programa, primero se prueban pequeños bloques de construcción del mismo, lo que normalmente se realiza con acceso al código fuente y con el soporte de herramientas de depuración, y a veces se implica a los desarrolladores que escribieron el código (Perry, 2006).

- Pruebas *de Integración*. Ven el sistema como una jerarquía de componentes donde cada uno pertenece a una capa de diseño. Tienen dos grandes objetivos: 1) detectar los errores que ocurren en las interfaces de unidades, y 2) ensamblar las unidades individuales en subsistemas de trabajo para obtener un micro-sistema completo para probar. Como consecuencia de esta integración, los micro-sistemas se ensamblan para ejecutar la prueba de integración (Burnstein, 2003).

- Pruebas *del sistema*. Se aplican al sistema completo, y tienen la tendencia de abordarlo desde el punto de vista funcional en lugar del estructural. Este nivel se considera normalmente como el más apropiado para comparar al sistema con sus requisitos no funcionales como seguridad, velocidad, exactitud y confiabilidad, además de las interconexiones externas con otras aplicaciones, utilidades, dispositivos hardware o con el sistema operativo (Hourizi & Johnson, 2001).

3.3 Técnicas para implementar pruebas funcionales

Las pruebas requieren técnicas o métodos para aplicar su metodología, que "son las encargadas de describir procesos para diseñar casos de prueba eficientes y eficaces en la localización de anomalías" (Mouelhi *et al*., 2009). Estas técnicas son un método, un camino o una vía que un probador utiliza para llegar a un fin: probar el software. Es necesario seleccionar la técnica más apropiada para cada situación de prueba específica, lo que equivale a decir que se debe seguir el camino que mejor conduzca a su objetivo (Beizer, 1990).

Las técnicas de prueba evolucionaron desde una actividad *ad hoc* hasta una disciplina organizada en la Ingeniería de Software; pero, aunque su madurez ha sido fructífera, no ha sido adecuada (Perry, 2006). Se espera que las técnicas de prueba ayuden al desarrollo de software de alta calidad, y a que se convierta en una de las áreas de investigación más importantes en el futuro próximo (Lou, 1990). Para las pruebas funcionales existen técnicas como tablas de decisión, partición de equivalencia, pruebas de valor límite, pruebas aleatorias, casos de prueba desde los casos de uso, entre otras.

- *Tablas de decisión*. Representan relaciones lógicas entre condiciones –mayoritariamente entradas– y acciones –mayoritariamente salidas. Los casos de prueba se derivan sistemáticamente considerando cada combinación de condiciones y acciones posibles. Cuando se utiliza una tabla de decisión para identificar casos de prueba, la propiedad de completitud de dicha tabla garantiza una prueba completa. No hay un orden particular para las condiciones ni para las acciones seleccionadas.

- *Partición de equivalencia*. Divide los datos de entrada de una unidad de software en una partición de datos, desde la que se puede derivar los casos de prueba. En principio los casos de prueba se diseñan para cubrir cada partición por lo menos una vez. Esta técnica trata de definir casos de prueba que revelen clases de errores, reduciendo así el número a desarrollar.

- V*alor límite o de frontera*. Trabaja con los valores extremos o límites de un rango de datos de entrada, permisibles para verificar la existencia de una anomalía. Esta técnica complementa la de partición de equivalencia y su extensión son las pruebas de robustez, donde se seleccionan casos de prueba que se encuentran fuera del dominio de las variables de entrada de datos, para comprobar la robustez del programa con entradas de datos erróneos e inesperados.

- *Aleatorias*. Es una técnica en la que un programa o sistema se prueba mediante la selección, al azar, de algún subconjunto de valores de todos los posibles valores de entrada. Se considera una técnica no óptima porque tiene una baja probabilidad de detectar un alto número de errores; sin embargo, a veces descubre anomalías que otras técnicas de prueba estandarizadas no detectan, por lo que debe considerarse como un complemento para las otras técnicas.

- *Casos de prueba desde los casos de uso*. La presente investigación se centra en el estudio al estado del arte de esta técnica, que tiene como propósito diseñar casos de prueba directamente desde la especificación de los requisitos en casos de uso, previamente estructurados en una plantilla de documentación en lenguaje natural. El objetivo es verificar que el sistema cumple con las especificaciones funcionales documentadas en los casos de uso originales.

Los casos de uso generalmente se crean por desarrolladores para desarrolladores, pero tienen una gran cantidad de información útil para los probadores, aunque en la mayoría de ocasiones el nivel de detalle especificado en ellos no es suficiente (Copeland, 2004). Los casos de uso se deben crear mediante un proceso de inspección y formalización, y la norma básica para probar la

implementación es crear al menos un caso de prueba para el escenario principal, y por lo menos otro para cada extensión (Cockburn, 2000). El sistema en este momento no es confiable, lo mismo que su calidad (Copeland, 2000).

3.4 Propuestas para aplicar técnicas de pruebas funcionales

En las diferentes épocas de la historia de la verificación del software se propusieron técnicas de prueba que a la vez dieron origen a investigaciones que culminaron con diferentes formas de aplicarlas. Cabe recordar lo sucedido con las pruebas funcionales, que surgieron en los 80 como respuesta a las necesidades de incrementar la calidad del software, y a la inquietud de la época de hacer de las pruebas un proceso continuo y paralelo al ciclo de vida de los proyectos de desarrollo. Rápidamente se estructuraron técnicas que permitieron aplicar esa metodología, pero hacía falta cómo ponerlas en práctica, por lo que algunos investigadores se dieron a la tarea de realizar proyectos de investigación que condujeron a las primeras propuestas de aplicación de esas técnicas en las pruebas funcionales. Ejemplos de entonces se pueden citar las técnicas *Equivalence Partitioning* y *Boundary Value Testing,* propuestas por Bohem (1979), y para las cuales, en menos de un año, se diseñaron varias propuestas de aplicación (Juristo *et al*., 2004).

Las técnicas para implementar pruebas requieren propuestas de aplicación para llevarlas a la práctica. Mientras que una técnica de prueba se utiliza para diseñar y seleccionar los casos de prueba que se aplicarán de acuerdo con el tipo de prueba determinado, una propuesta de prueba resulta de procesos de investigación que buscan la mejor forma de aplicar esa técnica. Debido a que cada técnica debe respetar unas reglas a través de las que se determina si el diseño de la prueba cumple o no con la metodología, se requieren propuestas de aplicación para programar y automatizar en la práctica continua el proceso de la prueba (Myers, 2004).

REFERENCIAS

- Agarwal, B. B., Tayal S. P. & Gupta M. (2010). "Software engineering and testing". USA: Jones and Bartlett Publishers.
- Allott, S. (2007). "Testing Techniques: Are they of any Practical Use?" *Academic and Industrial Conference Practice and Research Techniques, MUTATION*. Cumberland Lodge, UK, pp. 35-39.
- Basili, V. R. & Perricone B. T. (1984). "Software Errors and Complexity: An Empirical Investigation". *Communications of ACM*, Vol. 27, No. 1, pp. 42-52.
- Basili, V. R. & Selby R. W. (1987). "Comparing the Effectiveness of Software Testing Strategies". *IEEE Transactions on Software Engineering*, Vol. 13, No. 12, pp. 1278-1296.
- Baudry, B., Fleurey F., Jezequel J-M. & Traon Y. L. (2002). "Automatic test cases optimization using a bacteriological adaptation model: Application to .NET components". *17th IEEE International Conference on Automated Software Engineering, ASE'02*. Edinburgh, UK, pp. 253-257.
- Beizer, B. (1990). "Software Testing Techniques". New York: Van Nostrand Reinhold.
- Beizer, B. (1995). "Black-Box Testing: Techniques for Functional Testing of Software and Systems". USA: Wiley.
- Boehm, B. W. (1973). "The high cost of software". *Symposium on High Cost of Software*. Monterey, California, pp. 27-40.
- Bohem, B. W. (1979). "Guidelines for verifying and validating software requirements and design specifications". *European conference on applied information technology of the international federation for information processing*. London, UK, pp. 711-719.
- Brown, J. R. & Lipow M. (1973). "The quantitative measurement of software safety and reliability". *TRW Report SDP-1176*. TRW Software Series.
- Burnstein, I. (2003). "Practical software testing. A process-oriented approach". New York: Springer-Verlag.
- Cockburn, A. (2000). "Writing Effective use cases". New York: Addison-Wesley.
- Copeland, L. (2004). "A practitioner's guide to software test design". London: Artech House.
- Denger, C. & Medina M. M. (2003). "Test case derived from requirement specification". *IESE-Report 033.03/E*. Kaiserslautern, Germany.
- Dijkstra, E. W., Hoare C. A. R. & Dahl O. J. (1972). "Structured programming". New York: ACM Classic Books Series.
- Firesmith, D. G. (1993). "Testing Object-Oriented Software". *Software Engineering Strategies*, Vol. 1, No. 5, pp. 15-35.
- Glass, R. L. (1981). "Persistent Software Errors". *IEEE Transactions on Software Engineering*, Vol. 7, No. 2, pp. 162-168.
- Goodenough, J. B. (1979). "A survey of program testing issues". In P. Wegner (Ed.) Research Directions in Software Technology. Massachusetts: MIT Press, pp. 316-340.
- Gutiérrez, R. J. J., Escalona C. M. J., Mejías R. M. & Torres J. V. (2004). "Comparative analysis of methodological proposes to systematic generation of system test cases from system requisites". *Workshop on Systems Testing and Validation*. Paris, France, pp. 151-160.
- Gutiérrez, R. J. J., Escalona C. M. J., Mejías R. M. & Torres J. V. (2006). "Generation of test cases from functional requirements: a survey". *Workshop on System Testing and Validation*, Vol. 4, No. 4, pp. 117-126.
- Gutiérrez, R. J. J., Escalona C. M. J., Mejías R. M. & Torres Z. A. H. (2007). "Using use case scenarios and operational variables for generating test objectives". *5th Workshop on Systems Testing and Validation*. Paris, France, pp. 23-32.
- Harrold, M. J., Gupta R. & Soffa M. L. (1993). "A Methodology for Controlling the Size of a Test Suite". *ACM Transactions on Software Engineering and Methodology*, Vol. 2, No. 3, pp. 270-285.
- Hartmann, J., Imoberdorf C. & Meisinger M. (2000). "UML-Based Integration Testing". *International Symposium on Software Testing and Analysis*. Portland, Oregon, pp. 60-70.
- Hetzel, W. C. (1988). "The Complete Guide to Software Testing". Massachusetts: Weley.
- Hollnagel, E. (1993). "The phenotype of erroneous actions". *International Journal of Man-Machine Studies*, Vol. 39, No. 1, pp. 1-32.
- Horowitz, E. (1975). "Practical Strategies for Developing Large Software Systems". Massachusetts: Addison-Wesley.
- Hourizi, R. & Johnson P. (2001). "Unmasking Mode Errors: a new application of task knowledge principles to the knowledge gaps in cockpit design". *Proceedings of Human Computer Interaction, Interact'01*. Tokyo, Japan, pp. 255-262.
- Huffman, J. H. (1994). "Testing of Object-Oriented Programming Systems (OOPS): A Fault-Based Approach". *Lectures Notes in Computer Science*, Vol. 858, pp. 205-220.

- IEEE (1990). “Glossary of Software Engineering Terminology”. IEEE Standard 610.12.90.
- IEEE (1994). “Classification for Software Anomalies”. IEEE Standard 1044-1993.
- Jorgensen, P. C. (2004). “Software Testing: A Craftsman’s Approach”. New York: CRC Press.
- Juristo, N., Moreno A. M. & Vegas S. (2004). “Reviewing 25 Years of Testing Technique Experiments”. *Empirical Software Engineering*, Vol. 9, No. 1-2, pp. 7-44.
- Kamde, P. M., Nandavadekar V. D. & Pawar R.G. (2006). “Value of test cases in software testing”. *Management of Innovation and Technology*, Vol. 2, pp. 668-672.
- Kaner, C. (2003). “What is a good test case?” *StarEast conference 03*. Orlando, USA, pp. 34-50.
- Kaner, C., Falk J. & Nguyen H. Q. (1999). “Testing computer software”. New York: John Wiley & Sons.
- Kay, R. H. (2007). “The role of errors in learning computer software”. *Computers & Education*, Vol. 49, No. 2, pp. 441-459.
- Knuth, D. E. (1989). “The Errors of Tex”. *Software Practice and Experience*, Vol. 19, No. 7, pp. 607-685.
- Kobrosly, W. & Vassiliadiset V. (1988). “A Survey of Software Functional testing Techniques”. *IEEE S.T.T. Conference*. Binghamton, New York, USA, pp. 127-134.
- Lazonder, A. W. & Van Der Meij H. (1995). “Error-information in tutorial documentation: supporting user’s errors to facilitate initial skill learning”. *International Journal of Computer Studies*, Vol. 42, No. 2, pp. 185-206.
- Ledru, Y., du Bousquet L., Bontron P., Maury O., Oriat C. & Potet M. L. (2001). “Test purposes: adapting the notion of specification to testing”. *Automated Software Engineering. 16th IEEE International Conference on Automated Software Engineering, ASE'01*. San Diego, USA, pp. 127-134.
- Leszaka, M., Perryb D. E. & Stolla D. (2002). “Classification and evaluation of defects in a project retrospective”. *The Journal of Systems and Software*, Vol. 61, No. 3, pp. 173-187.
- Levendel, Y. H. (1990). “Reliability analysis of large software systems: Defect data modeling”. *IEEE Transactions on Software Engineering*, Vol. 16, No. 2, pp. 141-152.
- Lewis, W. E. (2009). “Software testing and continuous quality improvement”. USA: Auerbach Publications.
- Luo, L. (1990). “Software Testing Techniques: Technology Maturation and Research Strategy”. *Institute for Software Research International*. Carnegie Mellon University. Class Report for 17-939A. Pittsburgh, USA.
- Marick, B. (1990). “A Survey of Test Effectiveness and Cost Studies”. *Technical Report UIUCDCS-R-90-1652*, University of Illinois.
- Moore, J. W. (1994). “An integrated collection of software engineering standards”. *IEEE software*, Vol. 16, No. 6, pp. 51-57.
- Mouelhi, T., Traon Y. L. & Baudry B. (2009). “Transforming and Selecting Functional Test Cases for Security Policy Testing”. *2009 International Conference on Software Testing Verification and Validation*. Denver, Colorado, pp. 171-180.
- Myers, G. J. (2004). “The Art of Software Testing”. New York: John Wiley & Sons.
- Nebut, C., Fleurey F., Le Traon Y. & Jezequel J-M. (2004). “A Requirement-Based Approach to Test Product Families”. *Lecture Notes in Computer Science*, No. 3014, pp. 198-210.
- Ostrand, T. J. & Weyuker E. J. (1984). “Collecting and Categorizing Software Error Data in an Industrial Environment”. *Journal of Systems and Software*, Vol. 4, No. 4, pp. 289-300.
- Overbeck, J. (1993). “Testing Object-Oriented Software: State of the Art and Research Directions”. *First EuroSTAR Conference*. London, UK, pp. 5-25.
- Patton, R. (2000). “Software testing”. Indianapolis: Sams Publishing.
- Pérez, L. B. (2007). “Gestión de las Pruebas Funcionales”. *Actas de Talleres de Ingeniería del Software y Bases de Datos*, Vol. 1, No. 4, pp. 37-42.
- Perry, W. E. (2006). “Effective methods for software testing”. USA: Wiley Publishing Inc.
- Pfleeger, S. L. (2001). “Software Engineering: Theory and Practice”. USA: Prentice-Hall.
- Purchase, J. A & Winder R. L. (1991). “Debugging Tools for Object-Oriented Programming”. *Object-Oriented Programming*, Vol. 4, No. 3, pp. 10-27.
- Radatz, J. W. (1981). “Analysis of V&V data”. *Rome Air Development Center Report RADC-TR-81-145*. Logicon Inc. San Pedro, California.
- Raymond, S. E. (2001). “The cathedral and the bazaar: musings on Linux and Open Source by an Accidental Revolutionary”. USA: O'Reilly Media.
- Richardson, D. J. & Thompson M. C. (1988). “The RELAY model of error detection and its application”. *2nd Workshop on Software Testing, Verification and Analysis*. New York, NY, USA, pp. 223-230.
- Seo, K. I. & Choi E. M. (2006). “Comparison of five black-box testing methods for Object-Oriented software”. *Fourth International Conference on Software Engineering Research. Management and Applications*. Seattle, USA, pp. 213-220.
- Serna, M. E. (2010). “Métodos Formales e Ingeniería de Software”. *Revista Virtual Universidad Católica del Norte*, No. 30, Art. No. 7.

- Sinha, P. & Suri N. (1999). “Identification of Test Cases Using a Formal Approach”. *Twenty-Ninth Annual International Symposium on Fault-Tolerant Computing*. Madison, Wisconsin, pp. 314-322.
- Vegas, H. S. (2002). “Esquema de caracterización para la selección de técnicas de pruebas de software”. *Tesis doctoral*. Universidad Politécnica de Madrid.
- Vegas, H. S., Juristo N. and Basili V. (2006). “Packaging experiences for improving testing technique selection”. *Journal of Systems and Software*, Vol. 79, No. 11, pp. 1606-1618.
- Virvou, M. (1999). “Automatic reasoning and help about human errors in using an operating system”. *Interacting with Computers*, Vol. 11, No. 5, pp. 545-573.
- Wagner, S. (2004). “Efficiency Analysis of Defect-Detection Techniques”. *Technical Report TUMI-0413*, Institut für Informatik Technische. Universität München.
- Weiss, D. M. & Basili V. R. (1985). “Evaluating software development by analysis of changes: some data from the software engineering laboratory”. *IEEE Transactions on Software Engineering*, Vol. 11, No. 2, pp. 157-168.
- Weiss, D. M. (1979). “Evaluating Software Development from Error Analysis: The Data from the Architecture Research Facility”. *Journal of Systems and Software*, Vol. 1, pp. 57-70.
- Weitzenfeld, A. (2005). “Ingeniería de software orientada a objetos con UML, Java e Internet”. México: Thompson.
- Whittaker, J. A. (2000). “What is Software Testing? And Why Is It So Hard?” *IEEE Software*, Vol. 17, No. 1, pp. 70-79.
- Wolverton, R. W. (1974). “The cost of developing large scale software”. *IEEE Transactions on Computers*, Vol. 23, No. 6, pp. 615-636.

III. REVISIÓN DE LA LITERATURA

Desde la aparición de la técnica de prueba Casos de prueba desde los casos de uso a finales del siglo pasado, varios investigadores desarrollaron proyectos que terminaron en propuestas para aplicar en las pruebas funcionales. A continuación se describen las propuestas seleccionadas que conforman la muestra para este trabajo, y se aplican a un caso de estudio para analizar y comparar la eficiencia del conjunto de casos de prueba generado al seguir los pasos que cada una propone, para al final describir en un marco ontológico unificado los objetivos, conceptos y métodos de los enfoques del estado del arte

3.1 Estudio de caso

El estudio de caso utilizado se basa en el problema de circulación de material en una biblioteca. En el diagrama de la Fig. 3.1 y en la Tabla 3.1 se aprecia que existe dependencia de las interacciones *Prestar Material* y *Devolver Material* con *Validar Usuario*, de tal manera que no pueden ejecutarse antes que ésta; de igual manera existe una dependencia temporal entre las dos primeras, ya que *Prestar* debe haberse ejecutado antes que se ejecute *Devolver*.

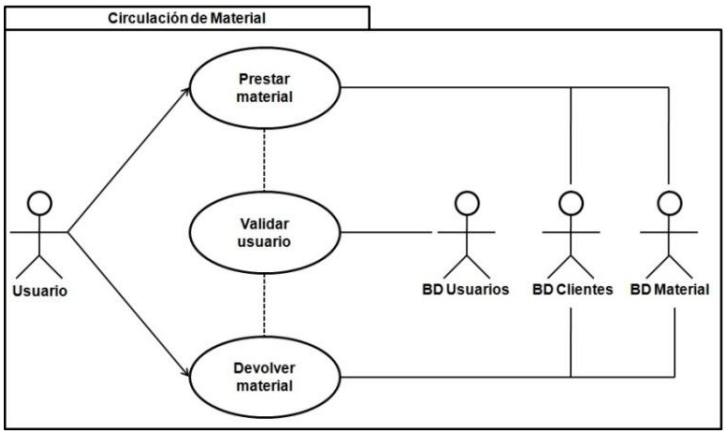

Fig. 3.1 Diagrama de casos de uso del estudio de caso

TABLA 3.1
Casos de uso del estudio de caso

Referencia	Descripción
CU-001	Validar Usuario
CU-002	Prestar Material
CU-003	Devolver Material

En las Tablas 3.2 a 3.4 se describe la documentación de las interacciones mediante la aplicación de la plantilla sugerida por Weitzenfeld (2004), y para que la documentación no se extienda innecesariamente se suprimen algunos elementos del modelo de la plantilla, considerados no necesarios en este estudio de caso.

TABLA 3.2
Interacción *Validar_Usuario*

Caso de uso	CU-001 Validar Usuario
Actores	Usuario, BD usuarios
Tipo	Inclusión
Propósito	Permitir la validación de usuarios en el sistema
Resumen	El Usuario carga la página inicial del sistema y digita nombre de usuario y contraseña
Precondiciones	Ninguna
Flujo Principal	1. El Usuario carga la página de acceso al sistema 2. El sistema despliega la página de verificación de datos de ingreso al sistema y solicita Usuario y Contraseña

	3. El Usuario digita los datos y selecciona la opción "Ingresar" 4. El sistema valida Usuario y Contraseña y despliega la página inicial del sistema
Excepciones	E1: Si el servidor no está activo o existen problemas de navegación, el sistema despliega mensaje de error y termina subproceso. E2: Si los datos de Usuario y Contraseña no se digitan, el sistema despliega mensaje de error y los solicita nuevamente. E3: Si los valores digitados para Usuario o Contraseña no se encuentran en la Base de Datos Usuarios, el sistema despliega mensaje de error y los solicita nuevamente. E4: Luego de tres intentos, el sistema despliega mensaje de error y termina subproceso

TABLA 3.3
Interacción *Prestar_Material*

Caso de uso	CU-002 Prestar Material
Actores	Usuario, BD Material, BD Clientes
Tipo	Principal
Propósito	Registrar el préstamo del material de la biblioteca
Resumen	El Usuario atiende las solicitudes de los clientes y las registra o rechaza de acuerdo con el escenario
Precondiciones	El Usuario debe estar validado en el sistema
Flujo Principal	1. El sistema presenta a Usuario cuadro de diálogo "*Prestar Material*" 2. El Usuario digita Código Cliente 3. El sistema valida Código Cliente en Base de Datos Clientes 4. El Usuario digita Código Material 5. El sistema valida Código Material 6. El sistema cambia estado de Cliente y Material, despliega mensaje de confirmación y termina subproceso
Excepciones	E1: Si Usuario no digita Código Cliente o Código Material, o son erróneos, el sistema despliega mensaje de error y los solicita nuevamente. E2: Si Código Cliente no está registrado o está suspendido, el sistema despliega mensaje informativo y termina subproceso. E3: Si Código Material no está disponible, el sistema despliega mensaje informativo y termina subproceso

TABLA 3.4
Interacción *Devolver_Material*

Caso de uso	CU-003 Devolver Material
Actores	Usuario, BD Material, BD Clientes
Tipo	Principal
Propósito	Registrar la devolución de material a la biblioteca por los clientes
Resumen	El Usuario atiende las devoluciones de material por los clientes
Precondiciones	El Usuario debe estar validado en el sistema
Flujo Principal	1. El sistema presenta a Usuario cuadro de diálogo "*Devolución Material*" 2. El Usuario digita Código Material 3. El sistema cambia estado de Cliente y Material, despliega mensaje informativo y termina subproceso
Excepciones	E1: Si Código material está en blanco o es incorrecto, el sistema despliega mensaje de error y lo solicita nuevamente. E2: Si Cliente incumple las normas de préstamo, el sistema cambio estado de Cliente a sancionado

3.2 Automated test case generation from dynamic models (Fröhlich & Link, 2000)

Esta propuesta parte de un caso de uso descrito en lenguaje natural anotado en una plantilla recomendada (Cockburn, 2000), y consta de dos bloques estructurales (Tabla 3.5).

TABLA 3.5
Pasos de *Automated test case generation from dynamic models*

Paso	Actividad	Objetivo
Bloque 1		
1	Traducir los casos de uso del lenguaje natural a los diagramas de estados.	Construir los diagramas de estados
Bloque 2		
1	Describir las pre y pos condiciones del diagrama mediante proposiciones	Diseñar un conjunto de proposiciones a partir del diagrama de estados
2	Definir las proposiciones que no dependen de los estados del diagrama	Diseñar el conjunto extendido de proposiciones
3	Generar las operaciones sobre el conjunto de extendido de proposiciones	Obtener el conjunto de posibles operaciones
4	Especificar los estados inicial y final del conjunto resultante de proposiciones	Diseñar los estados inicial y final del sistema mediante un conjunto de proposiciones

5	Definir cuál será el criterio de cobertura de la prueba	Estructurar el criterio de cobertura
6	Generar las pruebas mediante un algoritmo en lenguaje STRIPS	Generar el conjunto de casos de prueba

En el primer bloque traduce los casos de uso a diagramas de estados mediante reglas descritas (Fröhlich and Link, 1999); y en el segundo, a partir de los diagramas de estados, realiza las siguientes actividades:

1. Traducir las pre y pos-condiciones del diagrama a proposiciones
2. Extender la anterior actividad dado que pueden existir proposiciones que no dependen de los estados ni las transiciones del diagrama, para lo que anexa una proposición en la que el conjunto de datos es válido –está definido–, y otra en la que no es válido –no está definido–
3. Generar los operadores desde conjunto de proposiciones extendidas, proceso que consiste en una traducción sistemática utilizando técnicas de inteligencia artificial
4. Especificar los estados inicial y final del conjunto de proposiciones resultante –conjunto de requisitos, adicciones y sustracciones–
5. Definir con qué criterio se aplicará la cobertura de la prueba mediante un programa de inteligencia artificial, que toma las transiciones del diagrama de estados y, desde su estado inicial, analiza el posible estado final
6. Traducir el diagrama de estados, mediante la aplicación de un algoritmo, a lenguaje STRIPS (Fikes & Nilsson, 1971), y generar el conjunto de casos de prueba.

La propuesta presenta dos criterios para establecer la cobertura de las pruebas: considerar todos los estados o considerar todas las transiciones en el diagrama de estados. El producto final es un conjunto de transiciones en el diagrama de estados, expresadas con operadores, y sus proposiciones iniciales y finales, que se implementan en el sistema bajo prueba con el objetivo de comprobar si es posible obtener el resultado esperado. Los autores hacen las siguientes precisiones:

- Los diagramas UML, de estados, de acciones y de interacción, son los que mejor describen el comportamiento de un caso de uso, por lo que utilizan los diagramas de estados para definir alternativas simples y múltiples de ejecución, dado que facilitan la visualización de varios escenarios del caso de uso, y a que es posible modelar su comportamiento de forma más simple.
- Los elementos utilizadas en el proceso son: 1) un conjunto de requisitos o de proposiciones, que debe cumplirse para aplicar las operaciones; 2) adiciones, un conjunto de proposiciones agregadas al sistema cuando se ejecuta cada operación; y 3) sustracciones, proposiciones que se cancelan al ejecutar una operación.
- Traducir desde el diagrama de estados al conjunto de operaciones es un proceso sistemático en el que aplica el conjunto de reglas, que la misma propuesta tiene establecido, y que facilita la aplicación de técnicas de inteligencia artificial para el proceso de generación de los casos de prueba.

RESULTADO DE LA APLICACIÓN

Esta propuesta no permite generar casos de prueba que involucren secuencias de casos de uso, por lo que para este estudio de caso se aplica sólo a los casos de uso CU-001 y CU-002.

Bloque 1. Traducir los casos de uso del lenguaje natural a un diagrama de estados consiste. Para construir el diagrama se identifica el camino principal, en el que cada mensaje entre el sistema y los actores se convierte en una transición. Dado que el caso de uso 001 es una inclusión del 002, su diagrama de estados también lo es. El diagrama resultante se muestra en la Fig. 3.2.

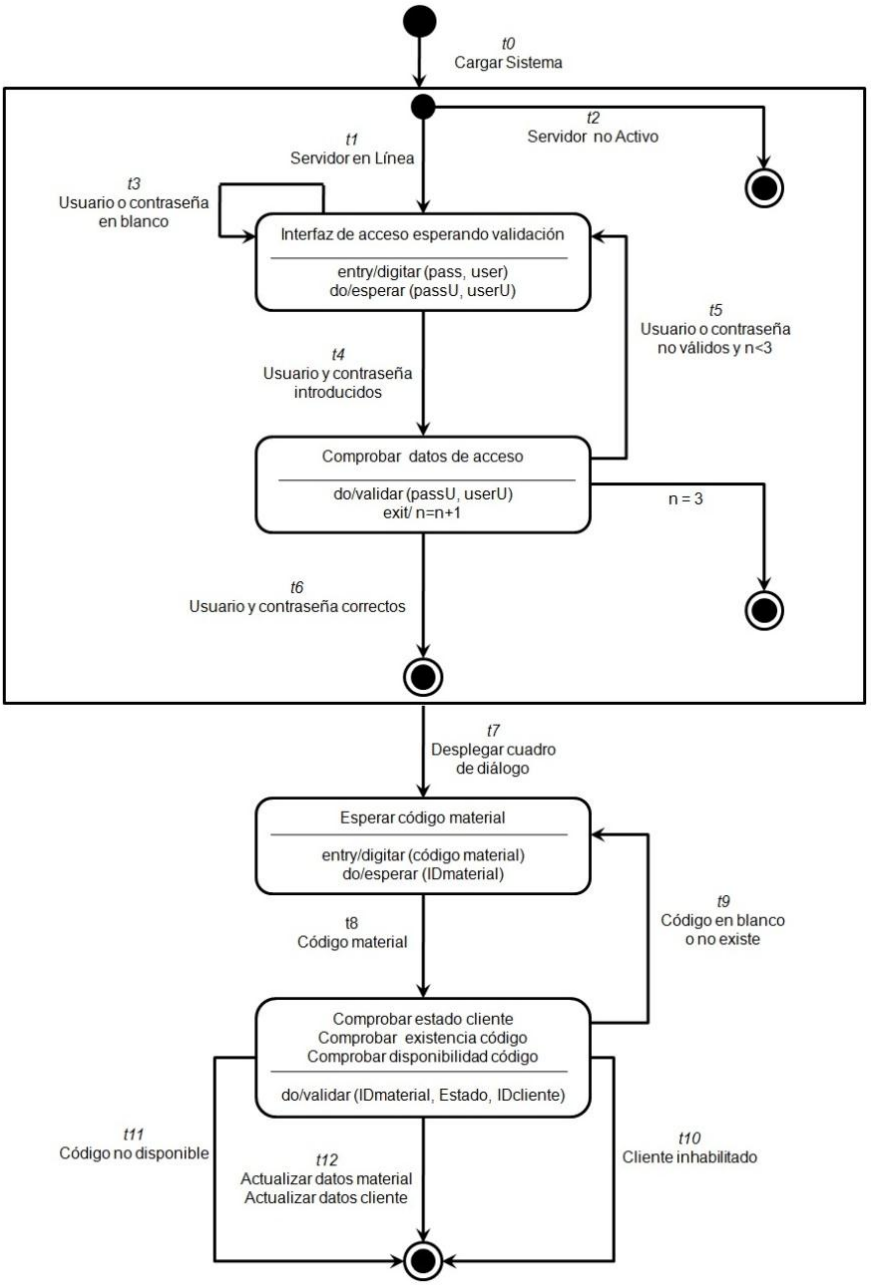

Fig. 3.2 Diagrama de estados de los casos de uso *Validar usuario* y *Prestar material*

Bloque 2

1. Extraer los operadores utilizando la notación del lenguaje STRIPS. De acuerdo con este lenguaje, un operador *O* es una terna de proposiciones: *O = {Pre, Add, Del}*. Donde *Pre* es el conjunto de proposiciones que debe existir para aplicarlo –precondiciones–, *Add* las nuevas proposiciones añadidas –proceso–, y *Del* las proposiciones eliminadas luego de su aplicación, como lo describe la Tabla 3.6. Para describir los estados del diagrama se utilizan los siguientes predicados auxiliares:

   - *In(t), t* representa el estado actual del sistema
   - *Log_state(t), t* representa el estado del sistema en algún momento pasado
   - *Log_trans(t),* representa que la transición *t* se utilizó en algún instante pasado; esta proposición se genera por cada transición en el diagrama de estados.

TABLA 3.6
Operadores resultantes

O	Pre	Add	Del
0	In(Inicial)	Log_state(Inicial) Log_trans(t0) In(e0)	In(Inicial)
1	In(e0)	Log_state(e0) Log_trans(t2) In(e0)	<vacío>
2	In(e0)	Log_state(e0) Log_trans(t1) In(e1)	In(s0)
3	In(e1)	Log_state(e1) Log_trans(t3) In(e1)	<vacío>
4	In(e1)	Log_state(e1) Log_trans(t4) In(e2)	In(e1)
5	In(e2)	Log_state(e2) Log_trans(t5) In(e1)	<vacío>
6	In(e2)	Log_state(e2) Log_trans(t6) In(e3)	In(e2)
7	In(e3)	Log_state(e3) Log_trans(t7) In(e4)	In(e3)
8	In(e4)	Log_state(e4) Log_trans(t8) In(e5)	In(e4)
9	In(e5)	Log_state(e5) Log_trans(t9) In(e4)	<vacío>
10	In(e5)	Log_state(e5) Log_trans(t10) In(final)	In(e5)
11	In(e5)	Log_state(e5) Log_trans(t11) In(final)	In(e5)
12	In(e5)	Log_state(e5) Log_trans(t12) In(final)	In(e5)

A este conjunto de operadores se aplica el algoritmo de planificación de la Tabla 3.7 –incluido en la propuesta– para generar las posibles secuencias del sistema a probar –casos de prueba.

TABLA 3.7
Algoritmo de planificación para *Automated test case generation from dynamic models*

Algoritmo	Crear serie de pruebas
Entrada	Una máquina de estados *M*
Salida	Una lista de secuencias de prueba y de limitaciones en los datos de prueba para cada secuencia
Proceso	▪ Generar el conjunto de definiciones de operador para *M* ▪ Crear un conjunto *A* con todas las proposiciones que ocurren en el diagrama de estados ▪ Para cada proposición $p \in A$: revisar si *p* ocurre en una sentencia de la forma: *Add:p* o *Del:p*. Si no, generar dos nuevos operadores para *p*, y registrarlos como restricciones de los datos de prueba: $\alpha_p^+ = (\emptyset, \{test\text{-}data(p), p\}, \emptyset\}$, $\alpha_p^- = (\emptyset, \{test\text{-}data(\sim p), \sim p\}, \emptyset\}$, Crear la definición de estado inicial: $\Sigma = U_{p \in A} \{\sim p\} \cup \{In(Inicial)\}$ ▪ Crear una lista de todas las transiciones y marcarlas como no cubiertas

Sub-diagramas	**IF** existen sub-diagramas: 1. Crear un contador para cada estado, indicando con qué frecuencia debe ocurrir en el conjunto de pruebas 2. Inicializar todos los contadores en 1 3. Para cada estado de referencia a un sub-diagrama ▪ Ejecutar este algoritmo para la sub-diagrama y guardar resultados ▪ Llevar número de secuencias de pruebas al contador de ocurrencias mínimas del estado de referencia del sub-diagrama.
Generar casos de prueba	***WHILE*** existan transiciones sin marcar ***OR*** un contador de ocurrencias > 0 para algún estado: 1. Seleccionar aleatoriamente una transición sin marcar o un estado con contador de ocurrencias > 0 2. Generar el objetivo de la definición para una secuencia de pruebas que cubra la transición o estado seleccionados 3. Invocar al planificador para obtener la secuencia de prueba 4. Marcar todas las transiciones cubiertas por el plan 5. Si existen, restarle al contador de ocurrencias de todos los estados el número de ocurrencias del estado en la secuencia de prueba 6. Obtener las limitaciones de los datos de prueba desde los seudo-operadores en el plan. Combinar las secuencias de prueba obtenidas con las formas de secuencia de los sub-diagramas. Restituir el conjunto de pruebas conformado por el conjunto de secuencias de prueba, junto con las limitaciones de los datos de prueba.

Luego de aplicar el algoritmo se encuentra una serie de secuencias –casos de prueba–, como las de la Tabla 3.8.

TABLA 3.8
Ejemplos de secuencias

Secuencia 1	Secuencia 2
Inicial	Inicial
e0	e0
e1	e1
e2	e2
e1	e3
e2	e4
e3	e5
e4	Final
e5	
Final	

3.3 Extended use case test design pattern (Binder, 2000)

La propuesta parte de un conjunto de casos de uso descritos en lenguaje natural y describe cómo extenderlos con información adicional para generar los casos de prueba. Aunque no detalla una estructura formal para representar los casos de uso, es necesario que éstos contengan un conjunto mínimo de información. El resultado final es una tabla que contiene todos los posibles casos de prueba para verificar todos los casos de uso. Se estructura en cuatro pasos como se observa en la Tabla 3.9.

TABLA 3.9
Pasos de *Extended use case test design pattern*

Paso	Actividad	Objetivo
1	Identificar las variables operacionales	Determinar la lista de variables operacionales para cada caso de uso
2	Definir el dominio de las variables operacionales	Encontrar el dominio de ejecución de cada una de las variables operacionales
3	Identificar las relaciones operacionales	Estructurar las tablas de decisión con las relaciones entre las variables operacionales
4	Diseñar y generar los casos de prueba	Encontrar el conjunto de casos de prueba que se va a aplicar al sistema

1. Tomar cada caso de uso y extenderlo con la siguiente información:
   - Las variables operacionales que describen el caso de uso –factores que varían de un escenario a otro–, y que determinan respuestas significativamente diferentes del sistema. Para cada una de estas variables se identifican los siguientes datos: valores de entradas y salidas, condiciones del sistema bajo las cuales la variable tendrá un comportamiento diferente al esperado, y una abstracción del estado del sistema que se prueba.
   - Las restricciones de dominio para cada variable operacional, es decir, el ámbito de acción de cada una.
   - La relación operacional, es decir definir claramente las secuencias operativas que pertenecen a cada uno de los casos de uso.
   - Opcionalmente, el valor de la frecuencia relativa de cada caso de uso.

2. Identificar el dominio de cada variable encontrada, es decir, los valores que la hacen válida o inválida.

3. Identificar las relaciones operacionales que determinan las dependencias entre las variables operacionales de los diferentes casos de uso, y expresarlas mediante tablas de decisión. Estas relaciones tienen asociadas las acciones del sistema que resultan cuando las variables toman un valor adecuado. Las tablas describen los posibles valores que deberá tener cada variable relacionada para originar la acción esperada.

4. Generar los casos de prueba desde la estructura de las tablas de decisiones del paso anterior; cada fila se convierte en un caso de prueba.

Por la cantidad de revisiones que sufren los casos de uso, y aunque es posible extender los casos de uso y generar los casos de prueba desde un primer momento, es conveniente posponer este proceso hasta que se logre: 1) que los casos de uso extendidos se desarrollen y validen, y 2) que el sistema bajo prueba haya pasado satisfactoriamente una serie de pruebas de integración, ya que con ese proceso es posible demostrar que los componentes tienen un mínimo de operatividad. Para determinar el grado de cobertura de las pruebas utiliza la fórmula descrita en Srivastava *et al*. (1997)

$$TC = \frac{Number\ of\ Implemented\ Capabilities}{Number\ of\ Required\ Capabilities} \times \frac{Total\ Components\ Tested}{Total\ Number\ of\ Components} \times 100$$

Y para el grado de cobertura de un caso de uso

$$XUCV = \frac{Number\ of\ Implemented\ UseCases}{Number\ of\ Required\ UseCases} \times \frac{Total\ Variants\ Tested}{Total\ Number\ of\ Variants} \times 100$$

RESULTADO DE LA APLICACIÓN

Esta propuesta tampoco tiene en cuenta las dependencias entre casos de uso, por lo que sólo se obtuvieron casos de prueba para el caso de uso CU-001. No explica cómo automatizar la generación de los casos de prueba, pero indica cómo automatizar su ejecución al proponer un proceso de grabación/reproducción de *scripts*. No define el número de casos de prueba a diseñar, pero sí cuando detener el proceso de generación.

1. Identificar las variables operacionales: Conexión al servidor, Nombre de usuario, y Contraseña de usuario

2. Definir el dominio de las variables. El dominio de ejecución de cada variable operacional se detalla en la Tabla 3.10.

TABLA 3.10
Dominio de las variables operacionales

	Variable operacional	**Domino**
1	Acceso servidor	Booleano: Si / No
2	Nombre usuario	Alfanumérico: Serie de caracteres/Cadena vacía
3	Contraseña usuario	Alfanumérico: Serie de caracteres/Cadena vacía

1. Identificar las relaciones operacionales. Incluyen todos los valores posibles de todas las variables operacionales. En la tabla 3.11 se presentan las relaciones operacionales para el estudio de caso.

TABLA 3.11
Relaciones operacionales encontradas

	Variables operacionales		
	Acceso sistema	Nombre usuario	Contraseña usuario
1	No	< >	< >
2	Si	Cadena vacía	Cadena vacía
3	SI	Cadena vacía	Serie de caracteres Contraseña incorrecta
4	Si	Cadena vacía	Serie de caracteres Contraseña correcta
5	Si	Serie de caracteres Nombre incorrecto	Cadena vacía
6	Si	Serie de caracteres Nombre correcto	Cadena vacía
7	Si	Serie de caracteres Nombre incorrecto	Serie de caracteres Contraseña incorrecta
8	Si	Serie de caracteres Nombre incorrecto	Serie de caracteres Contraseña correcta
9	Si	Serie de caracteres Nombre correcto	Serie de caracteres Contraseña incorrecta
10	Si	Serie de caracteres Nombre correcto	Serie de caracteres Contraseña correcta

3. Diseñar y generar los casos de prueba para cada relación operacional identificada. Este paso no lo describe la documentación de la propuesta, por lo que se supone una respuesta del sistema al ejecutar cada relación operacional, como se describe en la Tabla 3.12.

TABLA 3.12
Casos de prueba resultantes

	Variables operacionales			**Resultado esperado**
	Acceso sistema	Nombre usuario	Contraseña usuario	
1	No	< >	< >	Mensaje error
2	Si	Cadena vacía	Cadena vacía	Mensaje error IF n ≤ 3 Solicitar nombre y contraseña
3	SI	Cadena vacía	Serie de caracteres Contraseña incorrecta	Mensaje error IF n ≤ 3 Solicitar nombre y contraseña
4	Si	Cadena vacía	Serie de caracteres Contraseña correcta	Mensaje error IF n ≤ 3 Solicitar nombre y contraseña
5	Si	Serie de caracteres Nombre incorrecto	Cadena vacía	Mensaje error IF n ≤ 3 Solicitar nombre y contraseña
6	Si	Serie de caracteres Nombre correcto	Cadena vacía	Mensaje error IF n ≤ 3 Solicitar nombre y contraseña
7	Si	Serie de caracteres Nombre incorrecto	Serie de caracteres Contraseña incorrecta	Mensaje error IF n ≤ 3 Solicitar nombre y contraseña
8	Si	Serie de caracteres Nombre incorrecto	Serie de caracteres Contraseña correcta	Mensaje error IF n ≤ 3 Solicitar nombre y contraseña

9	Si	Serie de caracteres Nombre correcto	Serie de caracteres Contraseña incorrecta	Mensaje error IF n ≤ 3 Solicitar nombre y contraseña
10	Si	Serie de caracteres Nombre correcto	Serie de caracteres Contraseña correcta	Página inicial del proceso

3.4 Scenario-based validation and test of software –SCENT (Ryser & Glinz, 2000)

SCENT fue un proyecto del Requirements Engineering Research Group en Zurich que inició en 1997 y culminó en 2002; aunque la base estructural se publicó en 1999 (Ryser & Glinz) la propuesta definitiva se hizo en 2000. Tiene como base los escenarios, con los que elicita, valida y verifica los requisitos, desde los cuales desarrolla los casos de prueba. SCENT es una propuesta para elicitar, analizar y definir requisitos con los que luego crea escenarios estructurados –que validan los clientes o usuarios– y que deben formalizarse en diagramas de estados. Desde esos diagramas deriva los casos de prueba, que sirven de base para las pruebas de integración y del sistema.

Parte de un conjunto de requisitos formulados por el cliente e incluye su propio proceso para crear y refinar escenarios; el resultado final es un conjunto de escenarios refinados y de casos de prueba; éstos últimos utilizados para verificar si los escenarios se implementaron correctamente. Los casos de prueba se describen en tablas y en lenguaje natural. Se estructura en dos bloques: 1) creación y formalización de escenarios, y 2) generación de casos de prueba.

Para SCENT, *escenario* es un conjunto ordenado de interacciones entre socios, generalmente entre un sistema y un conjunto de actores externos a él. Puede comprender una secuencia concreta de pasos de interacción –instancias–, o un conjunto de pasos posibles de interacción –escenario tipo. *Caso de uso* es una secuencia de interacciones que surge de la relación del actor o actores con el sistema. Los pasos de esta propuesta se detallan en la Tabla 3.13.

TABLA 3.13
Pasos de *SCENT*

Paso	Actividad	Objetivo
1	Crear escenarios	Diseñar el conjunto de escenarios del sistema
2	Formalizar escenarios	Obtener los diagramas de estado
3	Generar casos de prueba	Diseñar el conjunto de casos de prueba

1. Para elicitar, crear y organizar escenarios a partir de los requisitos, la propuesta describe una serie de tareas:
   - Encontrar los actores que interactúan con el sistema
   - Encontrar los eventos externos al sistema
   - Definir las salidas del sistema
   - Definir el dominio del sistema
   - Detallar los resúmenes de los escenarios: escenarios tipo e instancias
   - Asignar prioridades a los escenarios
   - Pasar las instancias de los escenarios a escenarios tipo y crear una descripción detallada de los eventos de cada uno mediante flujos de acciones
   - Crear el diagrama resumen del sistema
   - Primera revisión por parte de usuarios, que aportan sus comentarios
   - Extender los escenarios desde los anteriores aportes y refinar su descripción mediante un flujo de ejecución
   - Modelar los flujos alternativos de ejecución
   - Extraer los escenarios abstractos
   - Depurar los escenarios utilizando requisitos no funcionales

- Revisar el diagrama que resume el sistema
- Segunda revisión formal por parte de usuarios, quienes validan los escenarios resultantes.

Al finalizar este paso se obtiene un conjunto de escenarios refinados, validados por usuarios, y estructurados en plantillas en lenguaje natural.

2. Formalizar los escenarios consiste en construir un diagrama de estados para cada uno. La propuesta define este paso como un proceso informal y creativo por parte de los ingenieros, lo que dificulta su automatización; además, no describe cómo realizarlo. El diseño de los diagramas de los escenarios se puede desarrollar al mismo tiempo que el paso anterior, al disponer de los primeros escenarios; posteriormente, complementa cada escenario con anotaciones de las precondiciones, los valores de entrada, los valores de salida, los rangos válidos y los requisitos no funcionales, para lo que la propuesta tiene su propia definición y que es una extensión a la notación de diagrama de estados de UML.

3. Para generar los casos de prueba es necesario:
   - Generar casos de prueba desde los diagramas de estados, es decir, a partir de las rutas transversales que recorren el diagrama –todos los nodos y los enlaces del diagrama. Como resultado se obtiene una tabla con los casos de prueba descritos en lenguaje natural, en la que cada línea constituye un caso de prueba, y cada columna las precondiciones, las entradas, las acciones del sistema y las salidas esperadas. La propuesta aclara que sólo es posible generar secuencias válidas del sistema, por lo que las no válidas se deben generar manualmente.
   - Generar casos de prueba adicionales y dependencias entre escenarios, lo que se logra recorriendo transversalmente el diagrama de dependencias con base en los requisitos no funcionales. De acuerdo con la propuesta esta es una tarea obligatoria pero no detalla cómo llevarla a cabo ni cómo definir los casos de prueba generados.
   - Integrar los diagramas de estados y los casos de prueba en nuevos diagramas; un paso que, según la propuesta, es opcional por lo que no explica cómo hacerlo.

Las ideas principales de esta propuesta son:

- Utilizar escenarios para el análisis de requisitos en las pruebas de sistema.
- Validar los escenarios mediante una revisión regular por los usuarios y finalmente formalizarlos, con el objetivo de encontrar las contradicciones, ambigüedades y omisiones en su descripción.
- Definir de forma sistemática cómo utilizar los escenarios para generar casos de prueba.

RESULTADO DE LA APLICACIÓN

1. Los escenarios se crean y describen en una estructura propuesta:

   Scenario Description
   Descriptive, Characteristic Scenario Information

   Scenario SC-001: El usuario se valida en el sistema

   Description, Purpose, Goal of Scenario in Global Context. El usuario digita valores de ingreso al sistema. El sistema verifica los datos y acepta o rechaza los valores.

   Actors. Usuario: Empleado de la biblioteca encargado de atender las solicitudes de los clientes. BD Usuarios: Base de datos con los usuarios registrados para utilizar el sistema.

Precondition. El computador debe tener acceso al sistema.

Postconditions. Success: El usuario queda validado y puede ejecutar sus funciones. *Failure*: El usuario no digita los datos correctos en los tres intentos. El sistema cancela la validación. *Triggering Event:* El usuario requiere iniciar sección de trabajo para atender al público.

Normal Flow, Success Scenario
1. El usuario carga la página de acceso al sistema
2. El sistema despliega la página de verificación de datos, solicita usuario y contraseña
3. El usuario digita los datos y selecciona la opción "Ingresar"
4. El sistema valida los datos y despliega la página inicial del proceso

Alternative Course of Action, Exceptions, Error Handling
1a. El servidor no está activo
1a.1 El usuario debe intentar luego. Salida del escenario
3a. Usuario y contraseña en blanco
3a.1 El sistema muestra mensaje de error. Incrementa contador de intentos en 1. Retorna al paso 2
3b. Usuario o contraseña inválidos
3b.1 El sistema muestra mensaje de error. Incrementa contador de intentos en 1. Retorna al paso 2
3c. Contador de intentos igual a 3
3c.1 El sistema termina el proceso. Salida del escenario.

Flow of actions
1. El usuario carga la página de acceso al sistema
   IF el servidor no está activo
   THEN existen problemas de conexión y el usuario debe intentar luego
   Salida del escenario
2. El sistema despliega la página de verificación de datos, solicita usuario y contraseña
3. El usuario digita los datos y selecciona la opción "Ingresar"
   IF el usuario y la contraseña están en blanco
   THEN el sistema muestra mensaje de error
   Incrementa contador de intentos en 1
   Retorna al paso 2

   IF el usuario y la contraseña son inválidos
   THEN el sistema muestra mensaje de error
   Incrementa contador de intentos en 1
   Retorna al paso 2

   IF contador de intentos = 3
   THEN el sistema termina el proceso
   Salida del escenario

Connection to and Integration with Other Model Elements/Models. Los caracteres de usuario y contraseña deben estar incluidos en la tabla ASCII

Open Questions/Known Problems. ¿Es posible que los datos para validación de usuario se puedan autocompletar? ¿Se puede acceder a la validación desde cualquier terminal de la red?

Test Planning, Test Cases
- Las variables *usuario* y *contraseña* son de tipo alfanumérico de mínimo 8 caracteres
- La variable *código del material* en circulación es alfanumérico y tiene un máximo de 15 caracteres
- El cliente se verifica con su número de identificación –entero-
- La base de datos *Usuario* contiene los datos históricos de todo el personal de la biblioteca
- La base de datos *Clientes* contiene los datos históricos de todos los clientes que pueden utilizar el servicio de circulación de material
- La base de datos *Material* contiene los datos históricos de todo los tipos de material que la biblioteca pone en circulación
- El material se presta hasta por 5 días sin importar el tipo de cliente
- Un cliente se deshabilita por incumplimiento de la norma anterior, hasta por 15 días

2. Para formalizar los escenarios se aplica sólo el primero y segundo punto del segundo bloque de la propuesta, ya que son los únicos obligatorios, como se muestra en la Fig. 3.3.

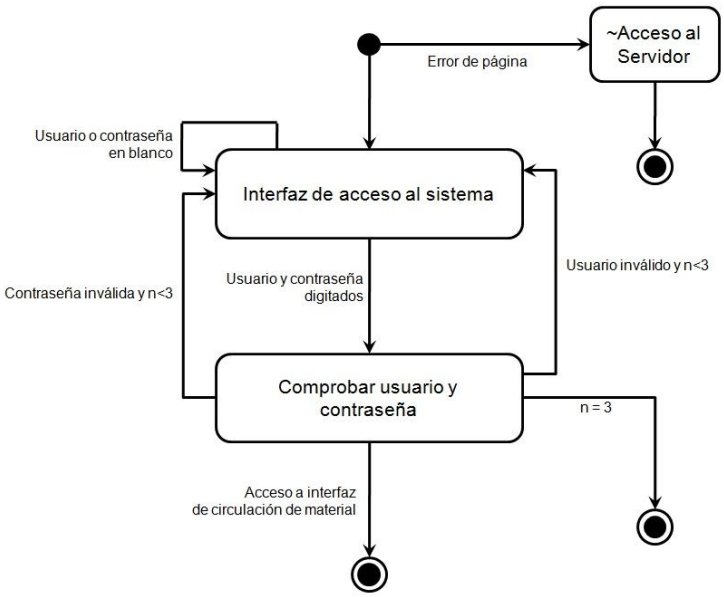

Fig. 3.3 Diagrama de estados del caso de uso *Validar usuario*

Luego se extiende este diagrama con las pre y post-condiciones, los valores de entrada, los datos de salida esperados, y los requisitos no funcionales, como se observa en la Fig. 3.4.

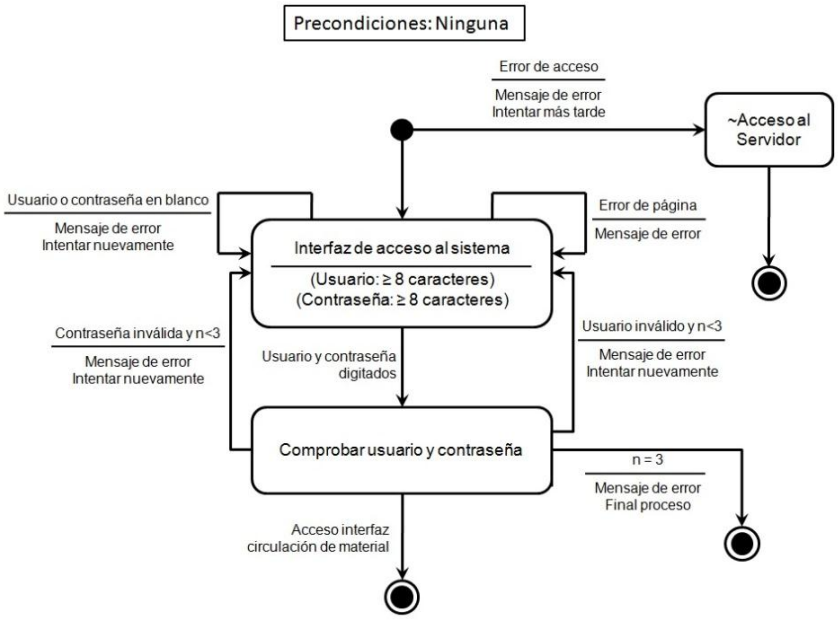

Fig. 3.4 Diagrama de estados extendido

3. Los casos de prueba se extraen desde los caminos que recorren el diagrama de estados. Para esto se localiza el camino principal como el primero y más importante de los casos de prueba, y luego los caminos alternativos de ejecución. Este proceso recorre todos los nodos y aristas del diagrama por lo menos con un caso de prueba. Para el estudio de caso, el caso de prueba principal sigue el camino de una ejecución normal: el usuario carga la interfaz de acceso al sistema, digita nombre de usuario y contraseña e ingresa a la interfaz de circulación de material. Todos los casos de prueba que se obtuvieron del diagrama de estados se describen en la Tabla 3.14.

TABLA 3.14
Casos de prueba obtenidos

Preparación de la prueba: Existe una base de datos de material para circulación y un usuario se valida exitosamente e ingresa al sistema			
Escenario	**Estado inicial**	**Desarrollo**	**Resultado esperado**
001	Interfaz de acceso al sistema	El actor digita un *usuario* y una *contraseña* válidos	Acceso interfaz de circulación de material
001	Interfaz de acceso al sistema	Existe un error de página	Mensaje error fallo de página
001	Interfaz de acceso al sistema	El actor digita un *usuario* o *contraseña* en blanco	Despliega mensaje de error y los solicita nuevamente
001	Interfaz de acceso al sistema	El actor digita un *usuario* de menos de 8 caracteres	Despliega mensaje de error y solicita nuevamente usuario y contraseña si n<3
001	Interfaz de acceso al sistema	El actor digita una *contraseña* de menos de 8 caracteres	Despliega mensaje de error y solicita nuevamente usuario y contraseña si n<3
001	Comprobar *usuario* y *contraseña*	Se valida correctamente *usuario* y *contraseña*	Acceso interfaz circulación de material
001	Comprobar *usuario* y *contraseña*	Se valida incorrectamente *usuario* y *contraseña*	Despliega mensaje de error y solicita nuevamente usuario y contraseña
001	Comprobar *usuario* y *contraseña*	El número de intentos de validación igual a 3	Despliega mensaje de error y regresa a interfaz de acceso al sistema
001	Error de acceso	Error de acceso	Mensaje de error

ESCENT asume que estos casos de prueba se pueden refinar adicionándoles el resto de la información descrita en los requisitos, y que al aplicar técnicas de prueba de dominio y de flujos de datos es posible derivar casos de prueba adicionales, pero no indica cómo hacerlo.

3.5 A UML-based approach to system testing –TOTEM (Briand & Labiche, 2001)
El punto de partida de esta propuesta es una serie de diagramas diseñados en la fase de análisis, y luego, desde ese modelo del sistema en notación UML, genera los casos de prueba. A partir de esos diagramas es posible obtener: requisitos, casos, oráculos y constructores de prueba. Los diagramas son:

- Diagramas de casos de uso descritos en lenguaje natural
- Diagramas de secuencia o colaboración
- Diagrama de clases
- Diccionario de datos
- Restricciones descritas en OCL

Esta propuesta está soportada en trabajos previos (Ince, 1992), (Beizer, 1999), (Binder, 2000), y detalla cómo utilizar los diagramas de actividades para capturar las dependencias secuenciales entre casos de uso, y permitir la especificación de secuencias de casos a probar. La cuestión clave para cada caso de uso que participa en una secuencia en particular consiste en seleccionar sus escenarios para someterlos a prueba, es decir, a cuáles caminos cubrir en los diagramas de secuencias correspondientes. También describe cómo derivar la información clave para determinar las condiciones iniciales del sistema para los escenarios de prueba, y sus oráculos de prueba correspondientes. Sus autores indican que esta metodología se justifica en función de su potencial para la automatización e implicaciones en términos de capacidad de prueba.

Para generar los casos de prueba propone ocho pasos, como se observa en la Fig. 3.5, de los cuales describe sólo tres estructurados en un proceso continuo. Cada paso se subdivide en varias tareas cuyo objetivo es obtener componentes y diagramas que, en conjunto, generan los casos de prueba.

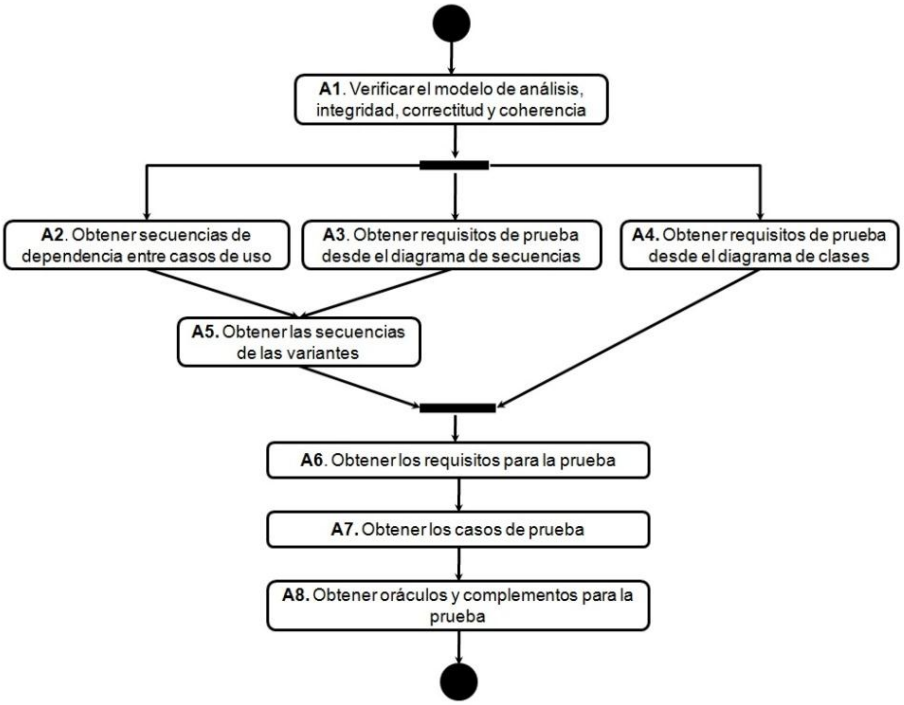

Fig. 3.5 Pasos de *TOTEM*

1. A2. *Derivar secuencias de dependencia entre casos de uso*. Consiste en identificar las dependencias secuenciales de los casos de uso, es decir, cuáles casos de uso dependen de otros para su ejecución. Las actividades son:
   - Representar las dependencias entre los casos de uso, y sus parámetros, mediante diagramas de actividades. La propuesta describe cómo diseñar un diagrama parametrizado por cada actor del sistema.
   - Generar las secuencias desde los diagramas de actividades y mediante un análisis con recorrido profundo a cada camino representado. Es conveniente realizar varios recorridos y analizar cada ruta en el diagrama, tarea que la propuesta deja a libertad del probador. Luego de obtener las secuencias generales, buscar combinaciones que permitan generar secuencias mayores, para lo que es necesario evitar la repetición de instancias en los casos de uso. Al final, generar los diagramas de secuencias o de colaboración.

2. A3. *Derivar requisitos de prueba desde los diagrama de secuencia*. Desde los diagramas del paso anterior se generan las secuencias de escenarios o de instancias de los casos de uso. La propuesta describe la realización de este paso a partir de la notación UML, en la que expresa todas las posibles alternativas que describe cada caso de uso. Las actividades son:
   - Expresar los diagramas de secuencias mediante expresiones regulares, que consiste en traducir la información de los diagramas de secuencias a expresiones regulares para obtener una expresión por cada diagrama. Cada secuencia de eventos del diagrama es una suma de productos en la expresión regular.
   - Identificar las condiciones para la realización de las rutas, es decir, obtener expresiones regulares compuestas de varios términos, donde cada uno representa una variante en la secuencia de interacciones del diagrama y tiene asociado un conjunto de condiciones en OCL que indican su realización o no realización.
   - Especificar las secuencias de operación desde las condiciones en OCL, y mediante la sustitución de los operadores por secuencias precisas en las expresiones regulares (*, -, +), especificar las secuencias de operaciones concretas que cada término ejecuta.

- Identificar los oráculos de prueba. *Test oracle* es un término no definido en esta propuesta, y que para esta tesis se asume como el concepto habitual manejado en entornos de pruebas (Binder, 2000). Se genera un oráculo por cada secuencia especificada a partir de las post-condiciones de una secuencia de interacciones.
- Construir las tablas de decisión. Estas tablas representan la formalización de toda la información obtenida hasta el momento; contienen los requisitos formales de la prueba y hacen parte integral del plan de pruebas; están conformadas por las condiciones iniciales de ejecución, que controlan las diferentes alternativas y acciones de las variantes; cada fila es una variante o instancia del caso de uso que los casos de prueba deberán cubrir al menos una vez.

3. A5. *Definir secuencias de variantes*. Este paso consiste en ejecutar las secuencias de operación y las tablas de decisión sobre secuencias de casos de uso, es decir, involucrar más de un caso de uso, aunque la propuesta no describe claramente cómo generar estas secuencias y sólo repite los anteriores pasos sobre más de un caso de uso a la vez.

RESULTADO DE LA APLICACIÓN

Los tres pasos que describe la documentación de esta propuesta son A2, A3 y A5. A1 comprueba que los diagramas UML están completos y que cumplan los requisitos de prueba; A4 no la aborda aunque es una contribución importante para probar los requisitos (Binder, 2000), pero lo propone como una línea de trabajo futuro. Sin embargo, las actividades A2, A3 y A5 constituyen en sí mismas, según sus autores, un componente autónomo de la propuesta, por lo que si falta A4 no se afecta la validez y utilidad de lo que presenta la documentación. Los pasos A7 y A8, utilizados para generar los casos de prueba y el código para los oráculos, son actividades que normalmente hacen parte de una fase posterior una vez esté disponible la información del diseño detallado.

Una vez garantizada la capacidad de la prueba –A1–, se obtienen los requisitos de prueba de los diferentes artefactos –A2 a A5–, que luego se fusionan en un conjunto de requisitos de prueba, evitando la redundancia y la combinación de los mismos en el plan de prueba. Por lo tanto, uno de los objetivos más importante de esta propuesta es proporcionar una metodología sistemática para realizar las actividades descritas y para automatizarlas en la medida de lo posible.

1. A2. Para representar las dependencias entre los casos de uso y para complementarlos, se utiliza el diagrama de actividades de UML, al que se agregan algunos parámetros –uno para cada actor. La Fig. 3.6 detalla el diagrama resultante en el caso de estudio. De este diagrama se generan las secuencias válidas de los casos de uso, que corresponden a la ejecución de un conjunto de ellos, y que son tantas como posibles combinaciones existan. De la lectura del diagrama se observan secuencias como las que muestra la Tabla 3.15.

TABLA 3.15
Secuencias generadas

	Secuencia
1	IngresarAlSistema(Usuario1Contraseña).PrestarMaterial(C1id, M1id)
2	IngresarAlSistema(Usuario1Contraseña).PrestarMaterial(C1id, M1id).DevolverMaterial(C1id, M1id)
3	IngresarAlSistema(Usuario2Contraseña).PrestarMaterial(C2id,M2id).PagarMulta(C2id,M2id). DevolverMaterial(C2id, M2id)
4	...

Además, debido a la diversidad de material y a que existen varios usuarios, los parámetros de entrada para las secuencias varían, por lo que la serie de repeticiones de cada una es decisión de los probadores. Luego, se combinan las secuencias entre sí, y todas las resultantes se

prueban en el sistema, como se observa en Tablas 3.16 y 3.17 y en la Fig. 3.7. Los autores especifican que todo este proceso se puede automatizar, pero en la documentación no encontramos ninguna herramienta referenciada para hacerlo, por lo que lo dejamos como actividad de trabajo futuro.

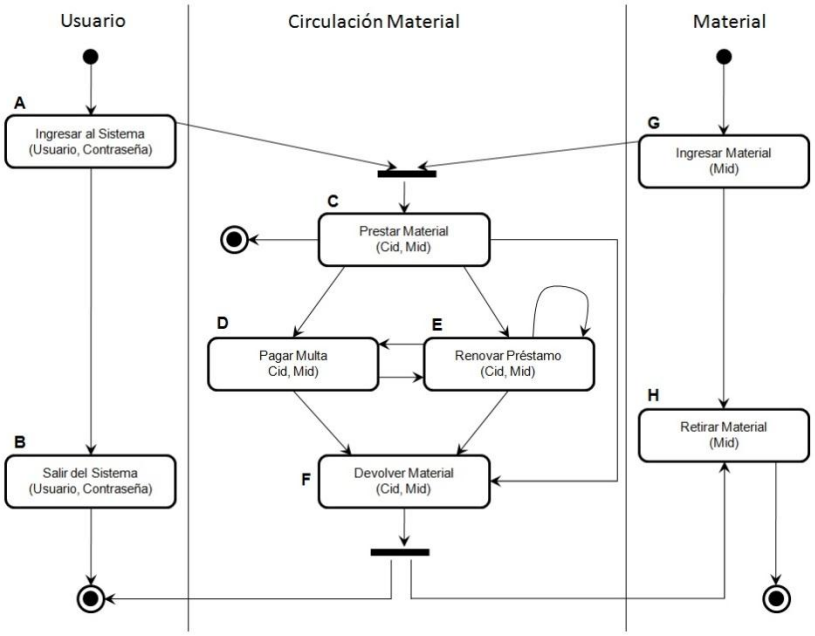

Fig. 3.6 Diagrama de actividades modificado

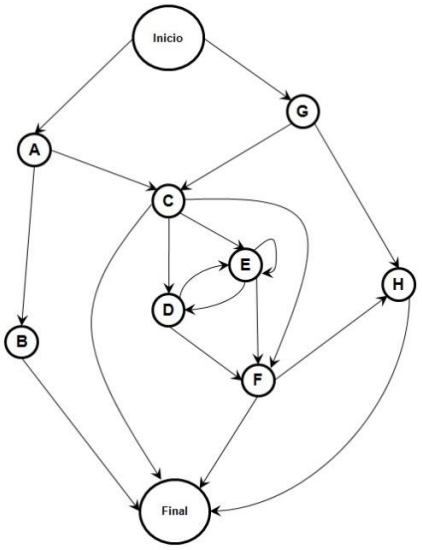

Fig. 3.7 Grafo de caminos resultante de las secuencias

TABLA 3.16
Secuencias combinadas

	Secuencia
1	IngresarAlSistema(Usuario1Contraseña).IngresarAlSistema(Usuario2Contraseña). PrestarMaterial(C1id,M1id).PrestarMaterial(C2id, M2id).
2	IngresarAlSistema(Usuario1Contraseña).IngresarAlSistema(Usuario2Contraseña). PrestarMaterial(C1id,M1id).PrestarMaterial(C2id, M2id).DevolverMaterial(C1id,M1id). DevolverMaterial(C2id, M2id)
3	...

TABLA 3.17
Secuencias parametrizadas

1. A (Usuario, Contraseña).B (Usuario, Contraseña)
2. A (Usuario, Contraseña).C (Cid, Mid)
3. A (Usuario, Contraseña).C (Cid, Mid).F (Cid, Mid)
4. A (Usuario, Contraseña).C (Cid, Mid).D (Cid, Mid).F (Cid, Mid)
5. …

2. A3. Para obtener los requisitos de prueba se ejecuta una tarea, independiente de los resultados obtenidos hasta ahora, para generar los casos de prueba para cada caso de uso. Se parte de un diagrama de secuencias en el que se tienen en cuenta todas las posibles interacciones entre sus clases, como se observa en la Fig. 3.8.

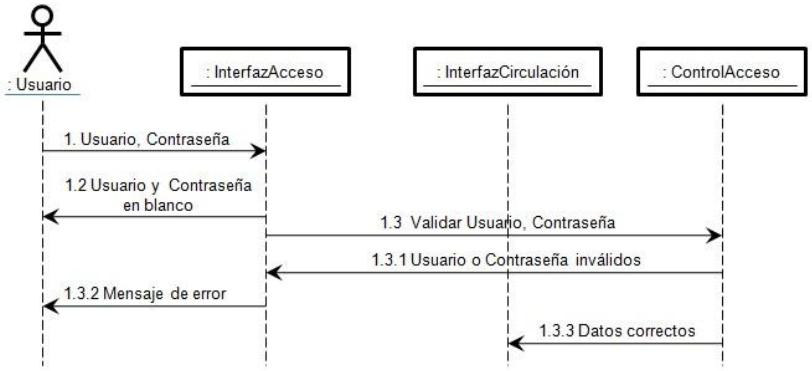

Fig. 3.8 Diagrama de secuencia de interacciones

Posteriormente se toma cada secuencia de este diagrama y se representa como una expresión regular, cuyo alfabeto son los métodos públicos de cada clase presente, como se observa en la Tabla 3.18.

TABLA 3.18
Expresiones regulares

Clase	**Secuencia**
InterfazAcceso	Digitar(Usuario, Contraseña)
InterfazAcceso	Usuario o Contraseña en blanco()
ControlAcceso	Validar(Usuario, Contraseña)
ControlAcceso	NoVálido(Usuario o Contraseña)
InterfazAcceso	MensajeError()
ControlAcceso	Válidos(Usuario y Contraseña)
InterfazCirculación	MensajeAccesoPermitido()

La propuesta plantea la siguiente notación para las expresiones regulares

Ingresar al sistema → $(\text{Digitar}_{\text{InterfazAcceso}}.\text{UsuarioOContraseñaEnBlanco}_{\text{InterfazAcceso}})^*$
Ingresar al sistema → $(\text{Digitar}_{\text{InterfazAcceso}}.\text{Validar}_{\text{ControlAcceso}}.\ \text{NoVálidos}_{\text{ControlAcceso}}.\text{MensajeError}()_{\text{InterfazAcceso}}.\text{Válidos}_{\text{ControlAcceso}}.$ $\text{MensajeAccesoPermitido}_{\text{InterfazCirculación}}$

Luego se suma los productos de las expresiones

Ingresar al sistema → $(\text{Digitar}_{\text{InterfazAcceso}}.\text{UsuarioOContraseñaEnBlanco}_{\text{InterfazAcceso}})^*$ + $(\text{Digitar}_{\text{InterfazAcceso}}.\text{Validar}_{\text{ControlAcceso}}.\ \text{NoVálidos}_{\text{ControlAcceso}}.\text{MensajeError}()_{\text{InterfazAcceso}}.\text{Válidos}_{\text{ControlAcceso}}.$ $\text{MensajeAccesoPermitido}_{\text{InterfazCirculación}}$

y se continua con la identificación de las condiciones necesarias bajo las que cada término se puede ejecutar –por tanto probar– y se expresan en OCL, como se presenta en la Tabla 3.19.

TABLA 3.19
Expresiones en OCL

	Expresiones regulares
A	Self.InterfazAcceso.Usuario → exists(Uid: Usuario | Uid=’’’’) and exists(Pass: Contraseña | Pass=’’’’)
B	Self.InterfazAcceso.Usuario → exists(Uid: Usuario | Uid<>’’’’) and exists(Pass: Contraseña | Pass<>’’’’) and (self. ControlAcceso.Válidos(Uid, Pass) == true)

3. A5. Obtener secuencias de las variantes es identificar las secuencias de operaciones que se ejecutaran para probar cada expresión regular, lo que se logra al expandir los operadores * y +. La decisión de cuándo parar las pruebas –número de repeticiones– queda a criterio de los probadores. En la Tabla 3.20 se detallan varias de las secuencias originadas de la variación del primer término de la suma de productos de las expresiones.

TABLA 3.20
Expresiones de la suma de productos

	Expresión regular
0	Ingresar al sistema → $Digitar_{InterfazAcceso}$.$Validar_{ControlAcceso}$.$Válidos_{ControlAcceso}$.$MensajeAccesoPermitido_{InterfazCirculación}$
1	Ingresar al sistema → $Digitar_{InterfazAcceso}$.$UsuarioOContraseñaEnBlanco._{InterfazAcceso}$.$Digitar_{InterfazAcceso}$. $Validar_{ControlAcceso}$.$Válidos_{ControlAcceso}$.MensajeAcceso $Permitido_{InterfazCirculación}$
2	Ingresar al sistema → $Digitar_{InterfazAcceso}$.$UsuarioOContraseñaEnBlanco._{InterfazAcceso}$.$Digitar_{InterfazAcceso}$. $UsuarioOContraseñaEnBlanco._{InterfazAcceso}$. $Digitar_{InterfazAcceso}$. $Validar_{ControlAcceso}$. $Válidos_{ControlAcceso}$.MensajeAcceso $Permitido_{InterfazCirculación}$

Por último, se estructuran las decisiones que aparecen en la Tabla 3.21.

TABLA 3.21
Tabla de decisiones

Variantes	**Condiciones**		**Acciones**		
CU-001	A	B	Mensaje I	Mensaje II	Cambio de estado
1	X	--	X	--	No
2	X	X	X	X	Si
3	--	X	--	X	Si

La propuesta no detalla cómo construir las tablas de decisión para secuencias de casos de uso, por lo que se aplica sólo a al caso de uso CU-001 de forma individual para el caso de estudio.

3.6 Generating test cases from use cases (Heumann, 2002)

Esta propuesta parte del principio de que el diseño de las pruebas es una tarea que se debe ejecutar desde las primeras etapas del ciclo de vida del producto, y describe cómo utilizar los casos de uso para generar los casos de prueba. Define al caso de uso textualmente en lenguaje natural y en una plantilla, y considera que su parte más importante son los caminos de ejecución. Estos caminos pueden tomar una de las siguientes direcciones: secuencia normal –principal–, camino que en condiciones normales sigue el sistema; o alternativa –excepciones–, las variaciones a causa de errores, rectificaciones, fallos o cualquier otra anomalía. A esos caminos le da el nombre de escenarios del caso de uso, y considera a todos los que aparezcan como base sobre la que diseña los casos de prueba. La propuesta se divide en tres pasos como se observa en la Tabla 3.22.

1. Generar los escenarios de prueba de los casos de uso. Donde identifica todas las posibles combinaciones entre la ruta principal de ejecución y las alternas y los detalla en una tabla.

2. Identificar el conjunto de casos de prueba para cada uno de los escenarios y condiciones de ejecución. Esta información también la enuncia en tablas pero sin notación o formalismo.
3. Identificar el conjunto de valores para cada caso de prueba.

TABLA 3.22
Pasos de *Generating test cases from use cases*

Paso	Actividad	Objetivo
1	Generar escenarios de prueba de los casos de uso	Encontrar el conjunto de escenarios de prueba por cada caso de uso
2	Identificar el conjunto de casos de prueba	Generar el conjunto de casos de prueba y las condiciones de ejecución para cada escenario
3	Identificar el conjunto de valores de prueba	Determinar el conjunto de valores de prueba que cada caso de prueba requiere

El resultado final es una tabla en la que describe, en lenguaje natural, todos los casos de prueba para verificar la correctitud de la implantación del caso de uso. Para su autor, la parte más importante de un caso de uso para generar casos de prueba es el flujo de eventos, y lo más importante de éste son el flujo básico y los flujos alternos. El primero procura porque el caso de uso se ejecute *normalmente*, mientras que los alternos cubren los comportamientos opcionales o excepcionales con respecto al comportamiento normal, así como las variaciones de dicho comportamiento. En la práctica, tiene en cuenta los casos de uso desde el comienzo hasta el final del ciclo de vida del desarrollo, mientras que en la parte final del mismo tiene en cuenta a los casos de prueba. Existen referencias a aplicaciones de esta propuesta en casos prácticos (Gutiérrez *et al*., 2006) (Gutiérrez *et al*., 2008).

RESULTADO DE LA APLICACIÓN

Un escenario es una instancia particular del caso de uso, y un caso de prueba es un conjunto de entradas, procesos de ejecución y salidas esperadas. Al aplicarla se generaron pruebas para verificar el comportamiento de los casos de uso aisladamente, ya que no tiene en cuenta las dependencias.

1. Para generar los escenarios de prueba desde los casos de uso se identifican todas las posibles combinaciones entre el camino principal y los alternativos, y se nombran; cada combinación resultante es un escenario de uso. La Tabla 3.23 contiene los escenarios de prueba para el caso de uso CU-001.

TABLA 3.23
Escenarios de prueba

	Escenario	Inicio	Excepciones
1	Fallo de servidor	Camino normal	Excepción 1
2	Acceso correcto al sistema	Camino normal	NA
3	Usuario y contraseña en blanco	Camino normal	Excepción 2
4	Usuario o contraseña inválidos	Camino normal	Excepción 3
5	Número de intentos igual a 3	Camino normal	Excepción 4

2. En este paso se obtuvieron las condiciones o valores necesarios para ejecutar cada escenario del paso anterior, como detalla la Tabla 4.20.

TABLA 3.24
Conjunto de casos de prueba

	Escenario	Proceso de ejecución			Salida esperada
		Fallo de servidor	Usuario	Contraseña	
1	Fallo del servidor	Si	NA	NA	Mensaje error
2	Acceso correcto al sistema	No	Válido	Válida	Página circulación de material

3	Usuario y contraseña en blanco	No	Vacío	Vacía	Mensaje error, n = n+1
4	Usuario o contraseña inválidos	No	No válido	NA	Mensaje error, n = n+1
5	Número de intentos igual a 3	No	NA	NA	Mensaje error. Página acceso

3. Para identificar el conjunto de valores de prueba, se revisa para todos los casos de prueba su exactitud, redundancia o ausencia; luego se identifican los valores de prueba para cada escenario de uso, como se muestra en la Tabla 3.25.

TABLA 3.25
Conjunto de valores de prueba

	Escenario	**Proceso de ejecución**			**Salida esperada**
		Fallo de servidor	**Usuario**	**Contraseña**	
1	Fallo del servidor	Error	NA	NA	Mensaje error
2	Acceso correcto al sistema	http ok	Diana	djana123	Página circulación de material
3	Usuario y contraseña en blanco	http ok	" "	" "	Mensaje error, n=n+1
4	Usuario o contraseña inválidos	http ok	Dianas	NA	Mensaje error, n=n+1
5	Número de intentos igual a 3	http ok	NA	NA	Mensaje error. Página acceso

Luego de repetir este proceso para cada caso de uso se obtiene una matriz con los casos de prueba requeridos. Con los escenarios de uso y los valores de prueba es posible realizar la prueba del sistema.

3.7 Testing from use cases using path analysis technique (Naresh, 2002)

La propuesta parte de un caso de uso descrito en lenguaje natural, desde el que elabora un diagrama de flujo con los posibles caminos para recorrerlo. Analiza esos caminos para asignarles una puntuación de acuerdo con su importancia y frecuencia; esos caminos también están descritos en lenguaje natural, sin ninguna presentación formalizada. Convierte a los caminos mejor evaluados en casos de prueba representados en un diagrama de flujo, y dado que de estos casos de prueba puede existir más uno probando al mismo tiempo un mismo camino, añade valores de prueba a cada uno para diferenciarlos. Tiene una estructura en cinco pasos, como representa la Tabla 3.26.

TABLA 3.26
Pasos de *Testing from use cases using path analysis technique*

Paso	**Actividades**	**Objetivo**
1	Elaborar diagramas de flujo desde los casos de uso	Diseñar los diagramas de flujo
2	Identificar en los diagramas los posibles caminos de ejecución	Determinar los caminos con los cuales recorrer los diagramas
3	Analizar los caminos y darles puntuación	Determinar la puntuación que puede asignarse a cada camino
4	Seleccionar los caminos mejor puntuados para probarlos	Hallar el listado de los caminos que se convertirán en casos de prueba
5	Elaborar los casos de prueba a partir de los caminos seleccionados	Estructurar un caso de prueba por cada camino seleccionado

Dado que para ejecutar cada caso de uso existen varios caminos, y a que cada uno es un potencial caso de prueba, es importante diseñar un diagrama de flujo que facilite su comprensión –describir una notación propia–, ya que es complicado analizarlos expresarlos en una plantilla de texto. En este diagrama se representa, con nodos, cada punto en el que el caso de uso puede seleccionar entre caminos, o el punto en el que convergen varios caminos luego de ejecutarse; y con ramas, a las acciones que el camino desarrolla. La Fig. 3.9 representa un ejemplo de diagrama de flujo de la propuesta.

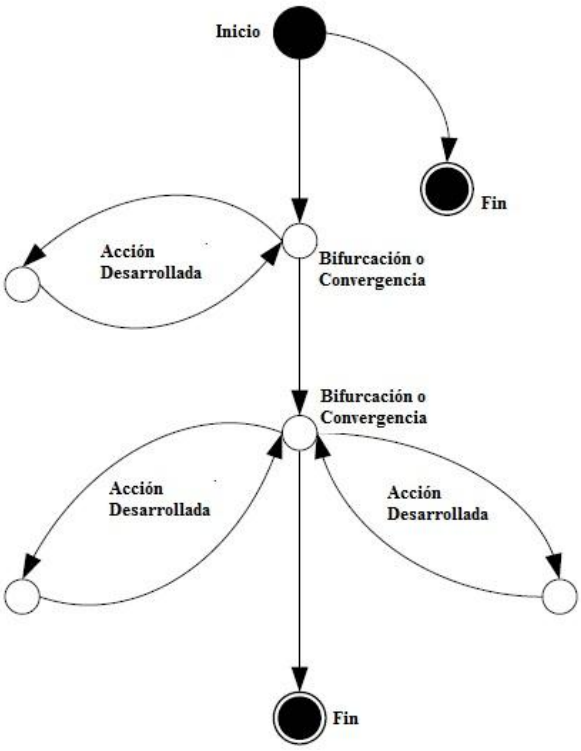

Fig. 3.9 Diagrama de flujo utilizado en *Testing from use cases using path analysis technique*

El siguiente paso consiste en construir un listado de camino del diagrama de flujo de ejecución del caso de uso. Le asigna a cada camino una serie de atributos valorados –la propuesta deja a discreción la cantidad de atributos a asignar y los valores de puntuación a colocar–, luego se obtiene una puntuación final para cada camino. La Tabla 3.27 representa un ejemplo.

TABLA 3.27
Puntuación ejemplo de *Testing from use cases using path analysis technique*

ID	Importancia	Frecuencia	Nombre	Descripción
1	10	1	2	Fallo de página
2	10	6	1, 4, 7	Acceso correcto al sistema

De acuerdo con la puntuación final se ordenan los caminos por importancia, luego se descartan los menos valorados para reducir el número de casos de prueba. Finalmente, antes de aplicar los caminos seleccionados como casos de prueba les asigna un valor a cada uno, ya que puede suceder que varios casos de prueba estén asignando puntuación diferente a un mismo camino. El autor concluye que analizar los caminos de los casos uso es una técnica extremadamente poderosa para crear casos de prueba; que los casos de prueba generados son realistas y son los que los usuarios encontrarán al utilizar el sistema; y que no sólo ahorra tiempo y dinero, sino que también ayuda en el control de riesgos.

RESULTADO DE LA APLICACIÓN

Como otras propuestas ya analizadas, sólo genera casos de prueba para casos de uso aislados debido a que no tiene en cuenta las dependencias con otros.

1. A partir de la secuencia normal de ejecución del caso de uso se dibuja el diagrama de flujo, en el que se enumera cada rama para facilitar la identificación de los caminos, como puede verse en la Fig. 3.10.

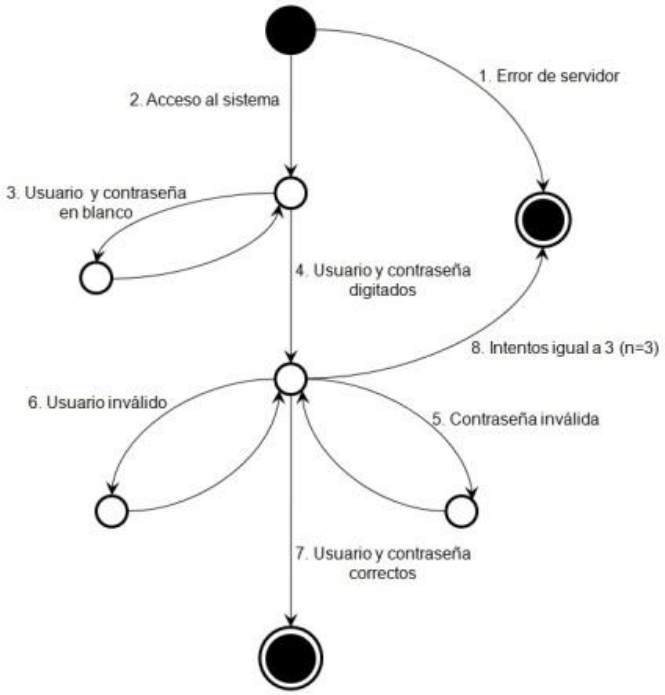

Fig. 3.10 Diagrama de flujo obtenido

2. Los posibles caminos de ejecución identificados se enumeran en la Tabla 3.28.

TABLA 3.28
Caminos de ejecución identificados

	Secuencia	**Descripción**
1	1	Error de acceso al servidor
2	2, 4, 7	Ingreso correcto al sistema
3	2, 3, 4, 7	Ingreso correcto al sistema luego de no digitar usuario ni contraseña
4	2, 4, 5, 7	Ingreso correcto al sistema luego de digitar una contraseña inválida
5	2, 4, 6, 7	Ingreso correcto al sistema luego de digitar un usuario inválido
6	2, 4, 5, 6, 7	Ingreso correcto al sistema luego de digitar una contraseña inválida y un usuario inválido
7	2, 3, 4, 5, 6, 7	Ingreso correcto al sistema luego de no digitar usuario ni contraseña, una contraseña inválida y un usuario inválido
8	2, 4, 8	Salida del sistema luego de tres intentos de validación (n=3)

3. Para la valoración de los caminos se utilizan los dos factores que describe la propuesta: frecuencia de ejecución y factor crítico, y se les asigna un valor entre 1 y 10. La recomendación que la propuesta hace para valorar la frecuencia es: 1 significa que el camino tiene una frecuencia de ejecución muy baja, entre 5% y 10%; 5 significa una frecuencia media, dado que se ejecuta en cerca del 50% de los recorridos del caso de uso; y 10 significa que tiene una frecuencia alta, entre el 90% y el 95% de los recorridos. Para el factor crítico: 1 significa que un error en ese camino no inhabilita al sistema, y que es posible recuperarlo sin pérdida de información y con un consumo pequeño de recursos; 5 significa que un error es importante para el sistema pero que puede funcionar; y 10 significa que un error es crítico para el sistema y que puede generar bloqueos.

   Luego de valorarlos se calcula el factor de valoración de cada camino: Valoración = Frecuencia + Factor crítico. Los resultados se muestran en la Tabla 3.29.

TABLA 3.29
Factor de valoración de los caminos

	Secuencia	Frecuencia	Factor crítico	Valoración
1	1	3	1	4
2	2, 4, 7	8	5	13
3	2, 3, 4, 7	3	4	7
4	2, 4, 5, 7	5	5	10
5	2, 4, 6, 7	7	5	12
6	2, 4, 5, 6, 7	4	5	9
7	2, 3, 4, 5, 6, 7	2	5	7
8	2, 4, 8	3	4	7

Posteriormente, se ordenan los caminos de acuerdo con su valoración. El resultado se muestra en la Tabla 3.30.

TABLA 3.30
Caminos ordenados por el factor de valoración

	Secuencia	Frecuencia	Factor crítico	Valoración
2	2, 4, 7	8	5	13
5	2, 4, 6, 7	7	5	12
4	2, 4, 5, 7	5	5	10
6	2, 4, 5, 6, 7	4	5	9
7	2, 3, 4, 5, 6, 7	2	5	7
3	2, 3, 4, 7	2	5	7
8	2, 4, 8	3	4	7
1	1	3	1	4

Para el estudio de caso, el camino principal es el que recorre las ramas 2, 4, 7, cuyo factor crítico es el mayor de todos los identificados. En la Tabla 3.30, todos los caminos alternativos vuelven a éste, por lo que, dado que un error en ellos redunda en el principal, el factor crítico es igual para todos.

4. Aunque el paso de seleccionar los caminos mejor puntuados para probarlos no se describe con detalle en la propuesta, se asume que el proceso consiste en seleccionar los caminos con mayor valoración, entre los que siempre debe estar el camino principal, y tener en cuenta que es necesario probar cada rama con al menos un caso de prueba. El resultado se muestra en la Tabla 3.31.

TABLA 3.31
Caminos seleccionados

	Secuencia	Frecuencia	Factor crítico	Factor	Descripción
2	2, 4, 7	8	5	13	Ingreso correcto al sistema
5	2, 4, 6, 7	7	5	12	Ingreso correcto al sistema luego de digitar un usuario inválido
4	2, 4, 5, 7	5	5	10	Ingreso correcto al sistema luego de digitar una contraseña inválida
3	2, 3, 4, 7	2	5	7	Ingreso correcto al sistema luego de no digitar usuario ni contraseña
8	2, 4, 8	3	4	7	Salida del sistema luego de tres intentos de validación (n=3)
1	1	3	1	4	Error de acceso al servidor

5. Para escribir los casos de prueba se diseña un conjunto de pruebas para verificar los caminos seleccionados, de tal forma que se garantice la comprobación adecuada del caso de prueba. El conjunto de pruebas consiste en escenarios descritos en tablas, con datos que permiten alcanzar con éxito el caso de prueba, y también con los que debe fallar. Los datos de prueba para el camino 4 se detallan en la Tabla 3.32 –en cursiva los correctos.

TABLA 3.32
Casos de prueba generados

Usuario	Contraseña	Nota	Resultado esperado
juan *juan*	Juank *juanca018*	Usuario correcto y contraseña incorrecta. Reemplazar contraseña por una válida, n+ = 1	Acceso a la interfaz de circulación del usuario juan
Diana *diana*	dianas123 *diana24*	Usuario incorrecto y contraseña incorrecta. Reemplazar usuario y contraseña por válidas, n+ = 1	Acceso a la interfaz de circulación del usuario diana
Marcela *marcela*	marce10co *marce10co*	Usuario incorrecto y contraseña correcta. Reemplazar usuario y contraseña por válidas, n+ = 1	Acceso a la interfaz de circulación del usuario marcela

3.8 Use case derived test cases (Wood & Reis, 2002)

Esta propuesta parte de un caso de uso descrito en lenguaje natural, del que indica toda la información que debe contener. El resultado final es un conjunto de casos de prueba, también descritos en lenguaje natural, que definen las acciones y verificaciones que realiza. El procedimiento consiste en identificar, a partir del flujo de eventos de cada caso de uso, todos los caminos de ejecución, y posteriormente transformarlos en un caso de prueba, también descrito en lenguaje natural. Los pasos se describen en la Tabla 3.33.

TABLA 3.33
Pasos de *Use case derived test cases*

Paso	Actividades	Objetivo
1	Desarrollar los casos de uso desde los requisitos funcionales	Construir un diagrama de casos de uso con las características que denota la técnica
2	Diseñar los escenarios	Hallar todos los posibles caminos que puede tomar un caso de uso
3	Estructurar los casos de prueba	Describir en lenguaje natural los casos de prueba desde los escenarios

1. Desarrollar los casos de uso desde los requisitos funcionales de la especificación. La información que cada caso de uso debe contener es: nombre, breve descripción, requisitos de la especificación, pre y post-condiciones, y el flujo de eventos. Luego aplica las siguientes interacciones: 1) identificar y consolidar sus componentes, 2) determinar el flujo de eventos, consistente de una lista de excepciones y de interacciones de los actores con el sistema y viceversa. La pre-condición especifica el estado del sistema antes del inicio del caso de uso, y puede utilizarse igualmente en el caso de prueba; la post-condición es el estado del sistema luego de cada interacción con un actor, y puede utilizarse como criterio para determinar si pasa o no la prueba.

2. Diseñar los escenarios. Un escenario es un camino único a través del flujo de eventos del caso de uso, y generalmente existe un camino que no tiene excepciones –escenario del día feliz–, lo mismo que varios derivados de él.

3. Estructurar y aplicar los casos de prueba. Desde los escenarios se selecciona cada uno de los caminos y se recorren teniendo en cuenta las excepciones y los requisitos funcionales. Cada camino genera un caso de prueba para validar cuándo se cumple, lo mismo que cada una de las excepciones.

Para la propuesta, un caso de uso es como una transparencia en el que sólo importa las entradas, las salidas y la funcionalidad. Sus propósitos son: promover la comunicación, ayudar a comprender los requisitos y a encapsular los datos, centrarse en el qué y no el cómo, y proveer los prototipos de casos de prueba. Se utiliza para descubrir los objetos que podrán satisfacer los requisitos funcionales en el sistema, y para construir los escenarios que aseguren que pueden soportar la funcionalidad.

Debido a las diversas acepciones de lo que es probar el software, los autores determinan lo siguiente: 1) verificar es demostrar el desarrollo de la funcionalidad requerida, y 2) validar es demostrar que el software es robusto y libre de errores. Además, la validación se alcanza con las pruebas de unidad, y la verificación con las pruebas funcionales. Para este último proceso definen que: 1) un caso de prueba es una parte de la prueba general ejecutada para identificar la funcionalidad requerida, y 2) un escenario de prueba es parte de la prueba funcional llevada a cabo con una de las variantes de funcionamiento especificadas.

La propuesta determina que las pruebas y el desarrollo del sistema deben realizarse mediante un proceso paralelo, como el que describe la Fig. 3.11; y concluye que derivar casos de prueba desde los casos de uso garantiza la funcionalidad requerida del software, lo que refleja en el plan de pruebas. Al trabajar los casos de uso con la misma especificación, es más probable llegar a la prueba con las mismas expectativas del sistema. Cuando el organizador de la prueba se apropia de los casos de uso se garantiza la coordinación entre desarrolladores y probadores.

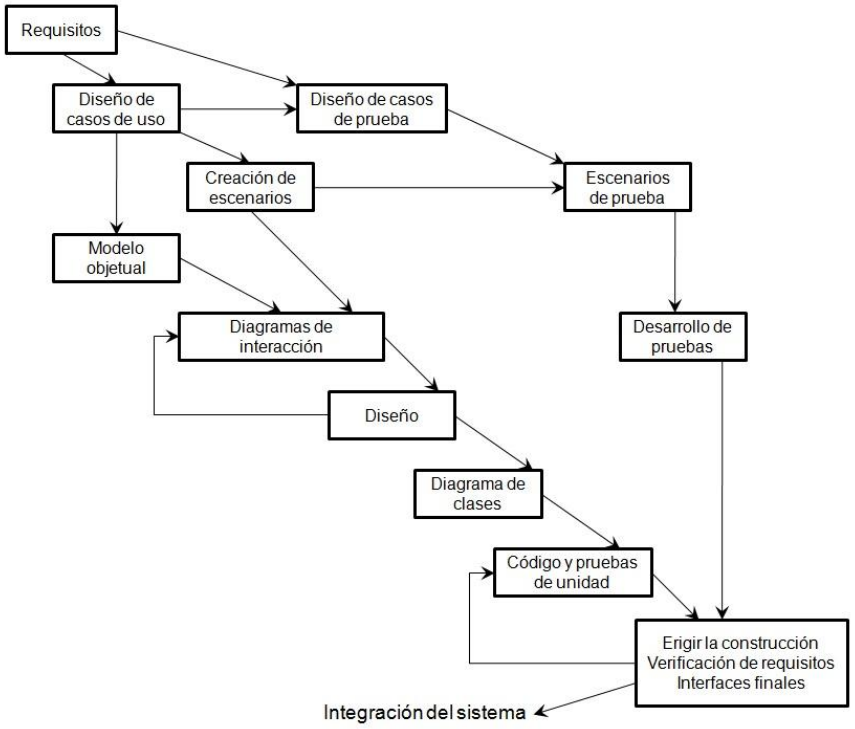

Fig. 3.11 Integración de las pruebas según *Use case derived test cases*

Los autores anotan que en su empresa, Harris Corporation, han utilizado esta metodología en múltiples proyectos con éxito, pero no ofrecen más detalles. Además, fue relacionada por Binder (2000), aunque su primera referencia es la de Wood & Reis (2002).

RESULTADO DE LA APLICACIÓN

Debido a que la documentación de esta propuesta no detalla cómo diseñar casos de prueba para casos de uso con dependencias, sólo se aplica al caso de uso CU-001.

1. El caso de uso estructurado es:
   - Nombre: CU-001 Validar Usuario
   - Descripción: Permitir la validación de usuarios en el sistema
   - Requisitos: Existe acceso al servidor, el Usuario debe estar registrado

- Pre-condiciones: Ninguna
- Post-condiciones: El Usuario queda validado y tiene acceso al servicio de circulación de material
- El flujo de eventos se detalla en la Tabla 3.34.

TABLA 3.34
Flujo de eventos resultantes

1. El usuario carga la página de acceso al sistema [E1] 2. El sistema despliega la página de verificación de datos de ingreso al sistema y solicita Usuario y Contraseña [E2] 3. El usuario digita los datos y selecciona la opción "Ingresar" [E3] 4. El sistema valida usuario y contraseña y despliega la página inicial del proceso [E3, E4]
▪ E1: El servidor no está activo o existen problemas de navegación, el sistema no carga la página de acceso, muestra un mensaje de error y termina subproceso ▪ E2: Si los datos de Usuario y Contraseña no se digitan, el sistema despliega un mensaje de error y los solicita nuevamente ▪ E3: Si Usuario o Contraseña no se encuentran en la Base de Datos Usuarios, el sistema despliega un mensaje de error y los solicita nuevamente ▪ E4: Luego de tres intentos, el sistema despliega mensaje de error y termina subproceso

2. El diseño de escenarios se ilustra en el Fig. 3.12.

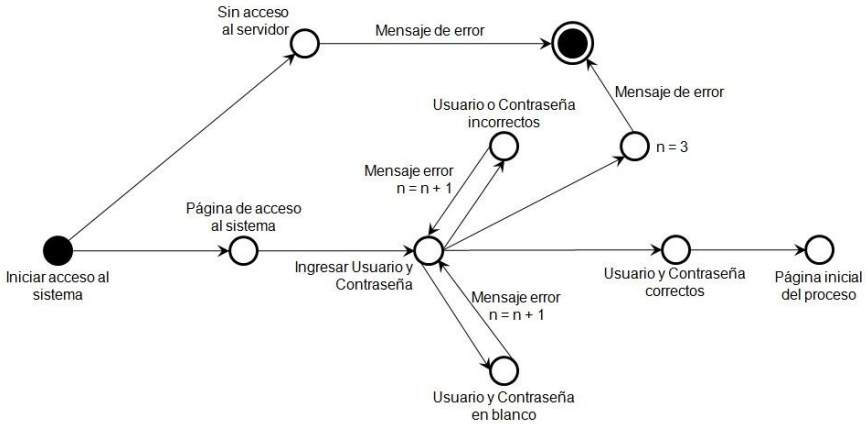

Fig. 3.12 Grafo de escenarios resultante

3. Los casos de prueba estructurados son:
   Caso de prueba 1. Escenario del día feliz.
   - El usuario digita Usuario y Contraseña correctos
   - El usuario inicia acceso al sistema
   - El sistema despliega interfaz de acceso al sistema
   - El usuario ingresa Usuario y Contraseña correctos
   - El sistema carga la página inicial del proceso

   Caso de prueba 2: no hay acceso al servidor.
   - El usuario inicia acceso al sistema
   - No hay acceso al servidor
   - El sistema despliega mensaje de error
   - Fin del caso de uso

   Caso de prueba 3: Usuario y Contraseña en blanco.
   - El usuario inicia acceso al sistema
   - El sistema despliega interfaz de acceso al sistema

- El usuario no ingresa ni Usuario ni Contraseña
- El sistema incrementa n en 1
- El sistema despliega mensaje de error
- El sistema solicita nuevamente Usuario y Contraseña

Caso de prueba 4: Usuario o Contraseña incorrectos.
- El usuario inicia acceso al sistema
- El sistema despliega interfaz de acceso al sistema
- El usuario ingresa Usuario o Contraseña incorrectos
- El sistema incrementa n en 1
- El sistema despliega mensaje de error
- El sistema solicita nuevamente Usuario y Contraseña

Caso de prueba 5: Número de intentos igual a tres (n=3).
- El usuario inicia acceso al sistema
- El sistema despliega interfaz de acceso al sistema
- Número de intentos de ingreso igual a tres
- El sistema despliega mensaje de error
- Fin del caso de uso

3.9 Requirements by contracts allow automated system testing (Nebut *et al.*, 2003)
Parte de un diagrama de casos de uso en notación UML, y al final proporciona el modelo de casos de uso extendido con contratos y el conjunto de casos de prueba, con los que verifica la implementación del modelo. Expresa los casos de uso mediante caminos de ejecución que recorren las secuencias para satisfacer pre y pos-condiciones. Los autores documentan los cuatro primeros pasos dela Tabla 3.35, en el último utilizan una herramienta para generar las pruebas.

TABLA 3.35
Pasos de *Requirements by contracts*

Paso	Detalle	Objetivo
1	Extender el modelo de casos de uso utilizando un lenguaje de contratos	Obtener casos de uso con parámetros y pre y pos-condiciones
2	Construir el modelo de ejecución de casos de uso	Diseñar el modelo de ejecución de casos de uso
3	Seleccionar el criterio de cobertura	Describir el criterio de cobertura para recorrer el modelo construido
4	Aplicar el criterio de cobertura seleccionado	Obtener secuencias de casos de uso válidas
5	Generar los casos de prueba	Diseñar las pruebas ejecutables para el sistema

La propuesta se estructura en dos partes, como se observa en la Fig. 3.13. En la primera extiende los casos de uso utilizando contratos, en los que incluye pre y pos-condiciones y los casos de uso con sus parámetros; y en la segunda describe cómo, a partir de los casos de uso extendidos, generar automáticamente los casos de prueba. En un trabajo posterior de los mismos autores (Nebut *et al.*, 2004) describen la aplicación de esta propuesta a familias de productos.

1. Extender el modelo de casos de uso. El objetivo es formular las dependencias que existe entre los casos de uso utilizando un intérprete o lenguaje de contratos que permita incluir parámetros y pre y pos-condiciones expresadas en proposiciones.
2. Construir el modelo de ejecución de casos de uso. Consiste en realizar un diagrama que permita expresar, a partir de los casos de uso extendidos, el comportamiento del sistema. En este diagrama, los nodos representan estados del sistema determinados por las proposiciones,

y las aristas representan las instancias del caso de uso. No existen restricciones para definir el significado semántico de los predicados.

3. Seleccionar el criterio de cobertura. Para este paso propone cuatro criterios para recorrer el modelo de ejecución, determinar el criterio de cobertura y obtener los casos de prueba: 1) de todas las aristas, por lo menos existe un objetivo de prueba por transición; 2) de todos los nodos, por lo menos existe un objetivo de prueba por nodo; 3) de todos los casos de uso instanciados, por lo menos existe un objetivo de prueba por caso de uso instanciado; y 4) de todas las aristas y casos de uso instanciados, por lo menos existe un objetivo de prueba por arista y caso de uso instanciado.
4. Aplicar el criterio de cobertura seleccionado. Al final se obtiene un conjunto de instancias de casos de uso con sus parámetros conformado por secuencias válidas de casos de uso.
5. Generar los casos de prueba. Mediante una herramienta, y a partir del conjunto de instancias del paso anterior, genera los casos de prueba.

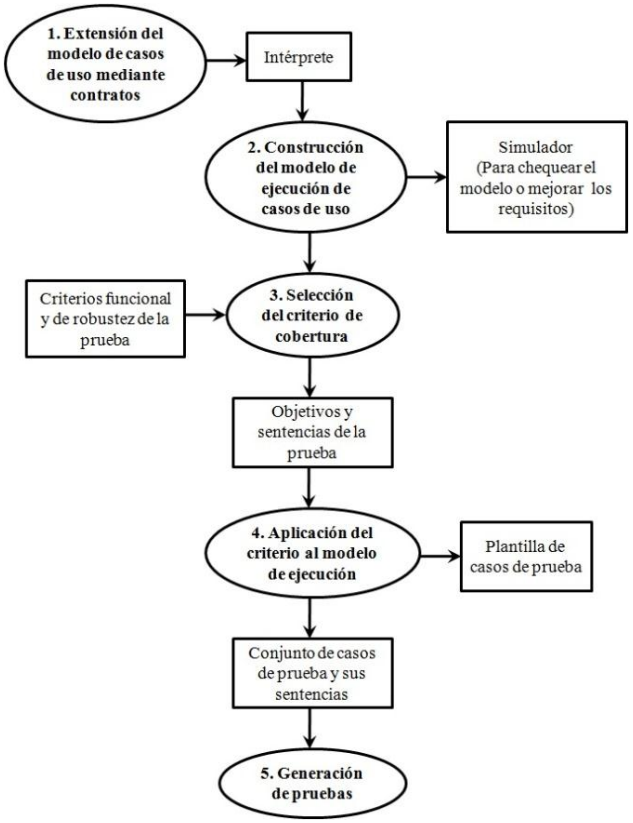

Fig. 3.13 Diagrama de actividades de *Requirements by contracts*

RESULTADO DE LA APLICACIÓN

1. La extensión del diagrama de casos de uso mediante contratos se muestra en la Tabla 3.36.

TABLA 3.36
Contratos para los casos de uso del sistema

Caso de uso	Función	Pre-condición	Post-condición
CU-001	Validar(u1)	Ninguna	UsuarioValidado(uv)
CU-002	Prestar(usuario uv)	UsuarioValidado(usuario uv)	MaterialPrestado(mp)
CU-003	Devolver(usuario uv, mp)	UsuarioValidado(usuario uv) y MaterialPrestado(mp)	MaterialDevuelto(md)

2. El modelo de ejecución de los casos de uso se detalla en la Tabla 3.37 y en la Fig. 3.14. De los parámetros y las pre y post-condiciones expresadas en los contratos de la Tabla 3.36, se extrae el orden de ejecución de los casos de uso. El modelo de ejecución detalla el progreso de los diferentes parámetros de los casos de uso; los estados detallan el estado del sistema de acuerdo con los predicados definidos en los contratos; y cada transición representa la ejecución de un caso de uso.

TABLA 3.37
Descripción de la semántica de los predicados

Predicado	**Detalle**
Validar(u1)	Determina que el usuario u1 debe quedar validado (uv) luego de ejecutar el caso de uso "Validar Usuario"
MaterialPrestado(mp)	Determina que el material mp se encuentra en estado préstamo
MaterialDevuelto(md)	Determina que el material md se encuentra en estado devuelto y puede prestarse nuevamente

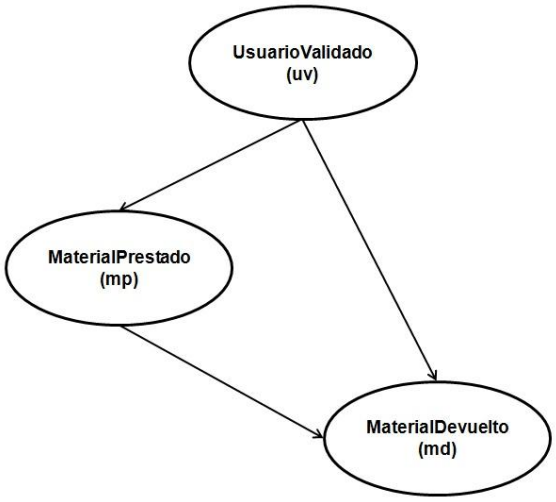

Fig. 3.14 Modelo de ejecución de los casos de uso

3. Se selecciona el criterio de cobertura de todos los estados y transiciones, por lo que se consideran todos los estados del modelo de la Fig. 3.14 en por lo menos un caso de prueba.

4. Para el paso de generar el conjunto de casos de prueba se utiliza la herramienta de libre distribución RBCTool. El modelo de ejecución de la Fig. 3.14 se estructura en la Tabla 3.38.

TABLA 3.38
Descripción del modelo de ejecución

```
# Entidades del sistema
{
        u1 : usuario
        md1, md2 : MaterialDisponible
}

# Descripción del estado inicial
{
        MaterialDisponible (md1)
        MateriaDisponible(md2)
}

# Descripción de los casos de uso
```

Caso de uso ValidarUsuario
CU Validar(u1: usuario)
pre
post UsuarioValidado(uv)
Caso de uso PrestarMaterial
CU Prestar(uv: UsuarioValidado; md : MaterialDisponible)
pre UsuarioValidado(uv) y MaterialDisponible(md)
post MaterialPrestado(mp) y MatrialNoDisponible(mp)

Caso de uso DevolverMaterial
CU Devolver(uv : UsuarioValidado; mp : MaterialPrestado)
pre usuarioValidado(uv) y MaterialPrestado(mp)
post MaterialDevuelto(md) y MaterialDisponible(md)

Las Tablas 3.39 a 3.42 representan las secuencias que obtuvimos luego de aplicar la herramienta al modelo de ejecución y con todos los criterios de cobertura de la propuesta.

TABLA 3.39
Criterio de todas las aristas

[ValidarUsuario(u1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv,md1), Devolver(uv,mp1)]
[Validadusuario(u1), Prestar(uv, md2), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md2), Prestar(uv, md1)]
[ValidarUsuario(u1), Prestar(uv, md2), Devolver(uv, mp2)]
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2), Devolver(uv, mp1)]
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2), Devolver(uv, mp2)]

TABLA 3.40
Criterio de todos los nodos

[ValidarUsuario(u1), Prestar(uv, md2)]
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2)]

TABLA 3.41
Criterio de todos los casos de uso instanciados

[ValidarUsuario(u1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), Devolver(uv,mp1)]
[ValidarUsuario(u1), Prestar(u1, md2), ValidarUsuario(u1)]
[Validarusuario(u1), Prestar(uv, md2), Devolver(uv, mp2)]
Lo que se debe cubrir:
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2), Devolver(uv,mp1), Devolver(uv, mp2)]

TABLA 3.42
Criterio de todos los nodos y casos de uso instanciados

[ValidarUsuario(u1), Validarusuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md1), Devolver(uv, mp1)]
[ValidarUsuario(u1), Prestar(uv, md2), ValidarUsuario(u1)]
[ValidarUsuario(u1), Prestar(uv, md2), Devolver(uv, mp2)]
[ValidarUsuario(u1), Prestar(uv, mp2)]
[Validarusuario(u1), Prestar(uv, md1), Prestar(uv, md2)]
Lo que se debe cubrir:
[ValidarUsuario(u1), Prestar(uv, md1), Prestar(uv, md2), Devolver(uv,md1), Devolver(uv, mp2)]

Nota: La herramienta no permitió la ejecución del proceso en el criterio de casos de uso instanciados, ni en el de todos los nodos y casos de uso instanciados, presenta un error de bloqueo interno y detiene el proceso de generación de casos de prueba.

3.10 Use case-based testing of product lines (Bertolino & Gnesi, 2003-2004)
El proceso para generar los casos de prueba que describe esta propuesta es una adaptación de lo que Ostrand & Balcer (1998) propusieron en *Category Partition,* pero la actualiza para trabajar con especificaciones modeladas con casos de uso para familias de productos. Describe cómo generar casos de prueba para familias de productos y parte desde los descritos en Bertolino *et al*. (2002), a los que extiende la notación con plantillas en lenguaje natural (Cockburn, 2000). En dichas plantillas describe los elementos comunes a la familia de productos, así como los puntos en los que cada producto se diferencia de los demás.

El proceso descrito consiste en generar descripciones abstractas de cada caso de prueba y luego refinarlas para diseñar casos de prueba ejecutables en el sistema, aunque en la documentación no describe cómo realizarlo. La propuesta se divide en ocho pasos, como se observa en la Tabla 3.43, los mismos de *Category Partition,* y el resultado final es un conjunto de casos de prueba comunes a todos los productos de la familia y uno específico para cada producto, descritos en plantillas en lenguaje natural.

TABLA 3.43
Pasos de *Use case-based testing of product lines*

Pasos	Actividades	Objetivos
1	Descomponer la especificación funcional del sistema en unidades funcionales	Hallar las unidades funcionales que sea posible probar independientemente
2	Identificar los parámetros y las condiciones del domino que interfieren con la ejecución de las unidades funcionales	Encontrar el conjunto de entradas y estado del sistema necesarios para ejecutar cada unidad funcional
3	Encontrar las categorías	Determinar una categoría para cada parámetro y condición del sistema e indicar el rango de valores que puede tomar
4	Dividir cada categoría en elecciones	Encontrar el conjunto de elecciones para cada categoría
5	Identificar las restricciones	Detallar la lista de las restricciones de cada elección
6	Redactar las especificaciones de prueba	Obtener el listado de las especificaciones de prueba
7	Generar el contexto de la prueba	Definir desde las especificaciones los marcos de prueba
8	Generar los scripts de prueba	Generar los conjuntos de scripts para aplicar en la prueba

1. Descomponer la especificación funcional del sistema en unidades funcionales. Su objetivo es hallar las unidades funcionales que se puedan probar de forma independiente. Este paso no lo aplica la propuesta ya que considera que las unidades funcionales son los casos de uso.
2. Identificar los parámetros y las condiciones del dominio que interfieren con la ejecución de las unidades funcionales. Tiene por objeto encontrar el conjunto de entradas y el estado del sistema, necesarios para ejecutar cada unidad funcional.
3. Encontrar las categorías. Encuentra una categoría por cada parámetro y condición del sistema con las que indica el rango de valores que pueden tomar.
4. Dividir cada categoría en elecciones. Con lo que halla el conjunto de elecciones para cada categoría.

   Los pasos 2, 3 y 4 los realizan los ingenieros de pruebas con base en su experiencia y conocimiento del sistema bajo prueba, y cada elección que realicen será un subconjunto de datos de cada categoría.

5. Identificar las restricciones. Su objetivo es determinar la lista de las restricciones para cada elección que expresa las dependencias entre ellas. Estas restricciones las describe con estructuras *IF* booleanas que asocia a cada elección. La restricción más utilizada es "*error*" que identifica las condiciones erróneas, y una elección con esta restricción no puede aplicarse en todas las combinaciones posibles para reducir el número de casos de prueba.

6. Redactar las especificaciones de prueba. Al final se obtiene el listado de las especificaciones de prueba, un documento estructurado de acuerdo con la plantilla de Ostrand & Balcer (1988), en el que recoge todas las categorías, las elecciones de cada una y sus restricciones.
7. Generar el contexto de la prueba. Su objetivo es definir, a partir de las especificaciones, los marcos de prueba a aplicar. Este proceso es automático a partir de las especificaciones de prueba. Un marco de prueba es una posible combinación de elecciones de todas las categorías que se encuentran en una especificación de prueba.

Repite los pasos 5, 6 y 7 las veces que sea necesario para refinar las especificaciones de la prueba.

8. Generar los *scripts* de prueba. Genera conjuntos de *scripts* que contienen varias de las pruebas generadas. Este paso consiste en traducir los marcos del paso anterior a un lenguaje ejecutable, para lo que los reúne aleatoriamente con el objetivo de diseñar los *scripts* de prueba que ejecutará.

La documentación de la propuesta no contempla los pasos 7 y 8, por lo que deben aplicarse tal como están descritos en *Category Partition*. Además, dado que esta propuesta tiene como base los requisitos estructurados y descritos en lenguaje natural, los casos de prueba derivados deben aplicarse de forma manual, particularmente la identificación de categorías relevantes y las opciones para la prueba. Sin embargo, es posible utilizar los analizadores léxicos y sintácticos de requisitos en lenguaje natural para extraer información útil que permita determinar las categorías pertinentes. En cambio, derivar los casos de prueba desde las especificaciones es una tarea fácilmente automatizable, y los autores anotan que están trabajando en una herramienta para automatizar la técnica al cien por ciento.

RESULTADO DE LA APLICACIÓN

1. Para realizar el primer paso se toman los casos de uso como unidades funcionales, por lo que cada caso de uso de la Fig. 3.1 será una unidad funcional.
2. No se precisa cómo identificar los parámetros y las condiciones del entorno de ejecución de cada unidad funcional, por lo que se debe recurrir a las características propias del problema, a la intuición y a la experiencia. Los parámetros y condiciones del sistema resultantes están descritos en la Tabla 3.44.

TABLA 3.44
Parámetros y condiciones del sistema

Predicado	Parámetros y condiciones
Validar Usuario	Estado Servidor Nombre Usuario Contraseña Usuario
Prestar Material	Código Cliente Estado Cliente Código Material Estado Material
Devolver Material	Código Cliente Código Material Estado Material

3. Cada parámetro y condición de la Tabla 3.44 debe tener su propia categoría, es decir, cada uno es una característica del sistema que se debe probar variando su valor.
4. Las elecciones indican los posibles rangos de valores para cada categoría. En la Tabla 3.45 se detallan las elecciones para cada categoría.

TABLA 3.45
Listado de elecciones

Categorías	Elecciones
Estado Servidor	Activo Inactivo
Nombre Usuario	Vacío Válido Inválido
Contraseña Usuario	Vacía Válida Inválida
Usuario	Validado No validado
Código Material	Vacío Válido Inválido
Estado Material	Disponible No Disponible
Código Cliente	Vacío Válido Inválido
Estado Cliente	Activo Sancionado Inactivo

5. Una restricción es una condición que debe cumplirse para que la categoría tenga un valor. Las mismas restricciones permiten la derivación del valor de una categoría desde el valor de otras. Se expresan estas restricciones, de acuerdo con lo propuesto por Ostrand & Balcer (1988), mediante sentencias *IF*. Las elecciones que requieren restricciones se detallan en la Tabla 3.47.

TABLA 3.47
Restricciones de las elecciones

Elección	Restricción
Usuario Validado	***IF*** Usuario válido ***AND*** Contraseña válida

6. Para las especificaciones de prueba se toma la información y se expresa en lenguaje TSL (Helmbold & Luckham, 1985) mediante una herramienta que automatiza el proceso; la propuesta no describe cómo generarlas ni referencia alguna herramienta para hacerlo.

Finalmente, y luego de aplicar los últimos pasos y siguiendo procesos lógicos de acuerdo con la experiencia, se obtienen los casos de prueba expresados en el lenguaje TSL. En la Tabla 3.47 se detallan los casos de prueba generados.

TABLA 3.47
Casos de prueba generados

Tag Nombre Usuario = Inválido *OR* Tag Contraseña Usuario = Invalida Ti: Scenarios: ext: Usuario No Validado
Tag Nombre Usuario = Válido *AND* Tag Contraseña Usuario = Valida Ti: Scenarios: ext: Usuario Validado

REFERENCIAS

- Beizer, B. (1990). "Software testing techniques". New York: International Thomson Computer Press.
- Bertolino, A., Fantechi A., Gnesi S., Lami G. and Maccari A. (2002). "Use case description of requirements for product lines". *The International Workshop on Requirements Engineering for Product Lines*. Essen, Germany, pp. 12-19.
- Bertolino, A. & Gnesi S. (2003). "Use Case-based testing of product lines". *ACM SIGSOFT Software Engineering Notes*, Vol. 28, No. 5, pp. 355-358.
- Bertolino, A. & Gnesi S. (2004). "PLUTO: A test methodology for product families". *Lecture Notes in Computer Science*, Vol. 30, No. 14, pp. 181-197.
- Binder, R. V. (2000). "Testing object-oriented systems: models, patterns and tools". New York: Addison-Wesley.
- Briand, L. & Labiche Y. (2001). "A UML-Based approach to system testing". *The 4th International Conference on the Unified Modeling Language, Concepts and Tools*. Toronto, Canada, pp. 194-208.
- Cockburn, A. (2000). "Writing Effective use cases". New York: Addison-Wesley.
- Copeland, L. (2004). "A practitioner's guide to software test design". Londres: Artech House.
- Fikes, R. E. & Nilsson N. J. (1971). "STRIPS: a new approach to the application of theorem proving to problem solving". *Artificial Intelligence*, Vol. 2, No. 3-4, pp. 189-208.
- Fröhlich, P. & Link J. (1999). "Modelling Dynamic Behaviour Based on Use Cases". *3rd International Software Quality Week Europe*. Brussels, Belgium, Paper 9A.
- Fröhlich, P. & Link J. (2000). "Automated test case generation from dynamic models". *Lecture Notes in Computer Science*, Vol. 1850, pp. 472-491.
- Gutiérrez, J. J., Escalona M. J., Mejías M. & Torres J. (2006). "An approach to generate test cases from use cases". *The 6th international conference on Web engineering*. Palo Alto, California, USA, pp. 113-114.
- Gutiérrez, J. J., Escalona M. J., Mejias M., Torres J. & Centeno A. H. (2008). "A Case Study for Generating Test Cases from Use Cases". *Second International Conference on Research Challenges in Information Science, RCIS'08*. Marrakech, Morocco, pp. 209-214.
- Heumann, J. (2002). "Generating Test Cases from Use Cases". *Journal of Software Testing Professionals*, Vol. 2, No. 3, pp. 24-35.
- Ince, D. (1992). "Object-Oriented Software Engineering with C++". New York: McGraw-Hill.
- Naresh A. (2002). "Testing from use cases using path analysis technique". *International Conference on Software Testing Analysis & Review, STARWEST'02*. Anaheim, California.
- Nebut, C., Fleurey F., Le Traon Y. & Jézéquel J-M. (2003). "Requirements by Contracts allow Automated System Testing". *Procedings of the 14th International symposium of Software Reliability Engineering, ISSRE'03*. Denver, Colorado, USA, pp. 85-96.
- Nebut, C., Fleurey F., Le Traon Y. & Jezequel J.-M. (2004). "A Requirement-Based Approach to Test Product Families". *Lecture Notes in Computer Science*, No. 3014, pp. 198-210.
- Ostrand, T. J. & Balcer M. J. (1988). "The category-partition method for specifying and generating functional tests". *Comunications of the ACM*, Vol. 31, No. 6, pp. 676-686.
- Ryser, J. & Glinz M. (1999). "A Practical Approach to Validating and Testing Software Systems Using Scenarios". *Third International Software Quality Week Europe*. Brussels, Belgium, Paper 12T.
- Ryser, J. & Glinz M. (2000). "Scent: A method employing scenarios to systematically derive test cases for system test". *Technical Report.* Institut für Informatik, Universität Zürich. IFI-2000.03.
- Srivastava, V. K., Farr W. & Ellis W. (1997). "Experience with the Use of Standard IEEE 982.1 on Software Programming". *International Conference and Workshop*. Monterey, CA, USA, pp. 121-127.
- Wood, D. & Reis J. (2002). "Use case derived test cases". *Software Quality Engineering for Software Testing Analysis and Review*. Memories, pp. 235-244.

IV. ANÁLISIS DE OBJETIVOS, CONCEPTOS Y MÉTODOS DE LAS PROPUESTAS

1. INDICADORES Y MÉTRICAS DE EVALUACIÓN

Los indicadores de eficiencia tienen como finalidad medir el nivel de ejecución de un proceso; se concentran en cómo se hacen las cosas, miden el rendimiento de los recursos utilizados en el mismo y, en términos generales, tienen que ver con la productividad. Teniendo en cuenta que para este trabajo se definieron tres características comunes a las propuestas seleccionadas: 1) partir de casos de uso, 2) descritos en lenguaje natural, y 3) ser de reciente promulgación, se determinó la siguiente metodología para definir los criterios de calidad y de eficiencia de los casos de prueba que se obtuvieron al aplicar cada propuesta:

1. Definir los atributos más importantes que deberían medir los indicadores.
2. Evaluar si los atributos poseen las características deseadas, es decir, ser medibles, entendibles y controlables.
3. Comparar los atributos evaluados con los propuestos en otras investigaciones, evitando redundancia o duplicidad.
4. Determinar el listado de indicadores que mejor responda a los atributos seleccionados.

Luego de discutir y complementar esta metodología se aplicó el siguiente procedimiento:

1. Definir atributos: tiempo de respuesta, cantidad de errores detectados, número de caminos recorridos, caminos repetidos, cobertura, operatividad, facilidad de uso, estabilidad, claridad de escenarios, definición de parada.
2. Evaluar atributos de acuerdo con las características, y determinar medibilidad, calaridad y controlabilidad, ya que su dominio se puede enmarcar en cantidades, procesos o resultados. Para lograrlo se estudiaron y discutieron las siguientes propuestas:
   - Boehm *et al.* (1978) propusieron el primer modelo para evaluar la calidad del software a través de métodos automáticos y cuantitativos aplicando una serie de atributos y métricas. Se trata de un modelo jerárquico basado en tres niveles: 1) usos principales, 2) componentes intermedios, y 3) características primitivas; los que utilizan indicadores como facilidad de uso, integridad, corrección, fiabilidad, eficiencia, facilidad de mantenimiento, facilidad de prueba, flexibilidad, reusabilidad, interoperabilidad, y portabilidad.

   - El modelo FURPS, presentado por Grady (1992) para clasificar cualidades de la calidad del software desde los requisitos, divide los factores principales de evaluación en dos grupos: 1) los basados en requisitos funcionales y 2) los basados en requisitos no funcionales. Los atributos que este modelo propone son:
     - **F**unctionality: características y capacidad del programa, generalidades de las funciones, seguridad del sistema.
     - **U**sability: factores humanos, factores estéticos, consistencia de la interfaz, documentación.
     - **R**eliability: frecuencia y severidad de los errores, exactitud de las salidas, tiempo medio de los errores, capacidad de recuperación, capacidad de predicción.
     - **P**erformance: velocidad de procesamiento, tiempo de respuesta, consumo de recursos, rendimiento efectivo total, eficacia.
     - **S**upportability: extensibilidad, adaptabilidad, capacidad de configuración, compatibilidad, capacidad de pruebas, compatibilidad, requisitos de instalación.

- El estándar ISO 9126 (2001), para la calidad externa de un producto software, distingue indicadores como efectividad, productividad, seguridad y satisfacción con la que un usuario interacciona con dicho producto en un ambiente y contexto específico; y para la calidad interna distingue funcionalidad, fiabilidad, usabilidad, eficiencia, mantenibilidad y portabilidad.

- La propuesta de James Bach, citado por Pressman (2004), para quien, en circunstancias ideales, un ingeniero de software debe diseñar un programa de computador, un sistema o un producto con la *facilidad de prueba* en mente, lo que le permite a los probadores diseñar casos de prueba con mayor facilidad. Los atributos sugeridos por Bach los puede emplear el ingeniero para desarrollar una configuración del sistema que se pueda probar: operatividad, observabilidad, controlabilidad, capacidad de descomposición, simplicidad, estabilidad, y facilidad de comprensión.

- Según Wagner (2004), los criterios de evaluación se definen con base en modelos funcionales del sistema, de manera que la calidad del conjunto de casos prueba será mejor cuantas más funcionalidades pruebe. Propone cuatro métricas para medir la eficiencia de detección de errores de las diferentes propuestas de pruebas: eficiencia de cobertura, eficiencia de cuenta de fallos, eficiencia de intensidad de fallos locales, y eficiencia de intensidad de errores por modelo. Las dos primeras se basan en las medidas de cobertura y en el número de errores encontrados, y las dos últimas en el tiempo como concepto de fiabilidad.

- El listado de atributos propuesto por Gutiérrez *et al*. (2005), quienes detallan una lista de atributos con su respectivo dominio y valor esperado: nueva notación, automatización completa, casos prácticos, tipo de pruebas, nivel de automatización, uso de estándares, automatización por herramientas, herramientas de soporte, dificultad de implantación, ejemplos, número de pasos, análisis y selección de casos de prueba.

3. Del análisis realizado a estas propuestas e investigaciones se seleccionaron los indicadores que se muestran en la Tabla 4.1 para aplicarlos en un primer análisis a las propuestas seleccionadas.

TABLA 4.1
Indicadores seleccionados para el análisis

Indicador	Descripción	Dominio
Documentación	Cuantifica el nivel de dificultad de aplicación de la propuesta a partir de la información disponible: Alta, si es posible aplicarla sólo con la documentación disponible; Media, si es posible aplicarla pero existirá lagunas o vacíos no resueltos; Baja, si es muy difícil aplicarla sólo con la documentación disponible	*Set{ Alta, Media, Baja}*
Casos prácticos	Si ofrece referencias a proyectos reales donde haya sido aplicada	*Boolean*
Construcción del modelo	Si incluye la construcción de un modelo de comportamiento del sistema o aborda directamente la generación de casos de prueba a partir de la información de los casos de uso	*Boolean*
Criterio de cobertura	Si detalla cómo determinar el criterio de cobertura	*Set{Enum{Análisis de caminos, Partición de categorías, Varios, No}}.*
Dependencias de casos de uso	Si puede generar casos de prueba que involucre más de un caso de uso o sólo puede generar pruebas a partir de casos de uso aislados	*Boolean*

Ejemplos de aplicación	Si incluye ejemplos además de casos prácticos	*Boolean*
Fecha	Año de divulgación. Se ha determinado que una propuesta requiere entre cuatro y seis años para que llegue a ser aceptada, discutida y soportada por la comunidad	*Entero*
Herramientas de soporte	Si existen herramientas que la soporten	*Boolean*
Momento de detención	Si describe en qué momento se detiene la generación de casos de prueba	*Boolean*
Momento de inicio	Si describe en qué momento se puede comenzar el proceso de generación de casos de prueba.	*Boolean*
Nueva notación	Si define una nueva notación gráfica	*Boolean*
Optimización de casos de prueba	Si describe cómo seleccionar un subconjunto de los casos de prueba generados	*Boolean*
Orden de los casos de prueba	Si describe el orden en que debe ejecutarse los casos de prueba	*Boolean*
Origen	Cuáles son los artefactos necesarios para aplicarla.	*Set{Necesidades, Lenguaje natural, Casos de uso, Varios}*
Pasos	Cantidad de pasos necesarios para su aplicación	*Entero*
Prioridad de casos de uso	Si es capaz de generar pruebas con mayor detalle para los requisitos o casos de uso más importantes, y generar pruebas menos detalladas para casos de usos secundarios	*Set{Alta, Media, Baja}*
Propuesta activa	Si los autores le dan continuidad a su trabajo o existe otros grupos que lo hacen; si existe herramientas de soporte que se estén desarrollando; y el nivel de referenciación	*Enum {Elicitación de requisitos, Análisis del sistema}*
Resultados	Si describe los resultados obtenidos luego de aplicarla a un estudio de caso	*Set{Enum{Lenguaje no formal, Lenguaje formal, Scripts de prueba, Modelo de prueba, Secuencia de transiciones}}*
Uso de estándares	Si utiliza diagramas y notaciones estándar o ampliamente aceptados	*Boolean*
Uso de notación gráfica	Si en algún paso utiliza diagramas gráficos	*Boolean*
Valores de prueba	Si detalla cómo seleccionar valores de entrada y salida para los casos de prueba	*Boolean*

2. ANÁLISIS DE CARACTERÍSTICAS E INDICADORES

2.1 Automated test case generation from dynamic models

Su principal ventaja es que describe y explica de forma sencilla el proceso de generación de casos de prueba, y que lo aplica en un ejemplo descriptivo, sin embargo, sólo fue posible generar casos de prueba a partir de los casos de uso aislados, ya que no tiene en cuenta sus dependencias. En la Tabla 4.2 se detallan las fortalezas y dificultades encontradas a esta propuesta.

TABLA 4.2
Fortalezas y dificultades de *Automated test case generation from dynamic models*

Fortalezas	**Dificultades**
▪ Detallar claramente todo el proceso para generar los casos de prueba ▪ Aplicar el proceso en un ejemplo descriptivo ▪ Aplicarse con cualquier herramienta de planificación con soporte en STRIPS ▪ Detallar en una guía cómo establecer la cobertura de la serie de casos de prueba que genera	▪ Tratar de manera aislada los casos de uso ▪ Para trabajar con casos de uso relacionados se deben incluir sub-diagramas de estados, lo que incrementa la complejidad de la técnica ▪ El proceso para generar los casos de uso está automatizado, mientras que traducir los casos de uso a diagramas de estados se debe hacer manualmente ▪ Traducir los diagramas de estados a operaciones es un proceso manual ▪ Esta traducción no puede hacerse totalmente de forma sistemática porque no es claro cómo incluir los operadores que no dependen de los diagramas –datos de prueba ▪ Al incrementar el conjunto de datos de prueba, o cuando éstos son complejos, también se incrementan las proposiciones, las operaciones y la complejidad del proceso

	▪ No es claro cómo implementar y aplicar los casos de prueba, ya que no ofrece una guía para hacerlo ▪ Dado que los casos de prueba se expresan en proposiciones, no es fácil implementarlos por lo que es necesario regresar y describir las operaciones

Al aplicar la propuesta se encontraron situaciones no documentadas y que no resuelve el ejemplo incluido, como qué hacer si existen varias transiciones que inician y finalizan en el mismo estado. No se encontraron herramientas libres basadas en STRIPS para implementar el modelo de comportamiento. Dado que la secuencia de posibles estados para implementar los casos de prueba sólo se obtuvieron a partir de la información de los mismos estados y operadores, se requirió un esfuerzo adicional para identificar los operadores ya aplicados en una determinada transición.

2.2 Extended use case test design pattern

El proceso de esta propuesta es un método más elaborado del descrito por Ostrand & Balcer (1988) de partición en categorías, en el que las variables operaciones corresponden a las categorías, los dominios a las elecciones, las relaciones operacionales a las restricciones, y así sucesivamente, y no aporta conceptos o procesos nuevos que éste no tenga. Una característica importante es que indica claramente el momento de inicio de las pruebas, y que tiene como base el estándar IEEE 982.1 (Srivastava *et al*., 1997) para evaluar la cobertura e índice de madurez; con dicha evaluación es posible decidir el momento de detener la generación de casos de prueba. La Tabla 4.3 contiene sus fortalezas y dificultades.

TABLA 4.3
Fortalezas y dificultades de *Extended use case test design pattern*

Fortalezas	Dificultades
▪ Indicar con claridad el momento de inicio y detención del proceso de la prueba ▪ Trabajar con el estándar IEEE 982.1 en la evaluación de la cobertura de la prueba	▪ No describir cómo se añade la información a un caso de uso extendido, ya que no tiene una notación, plantilla o extensión a UML ▪ Utilizar requisitos descritos en lenguaje natural, lo que hace difícil su sistematización y automatización ▪ No describir alguna referencia a aplicaciones reales ▪ No detallar cómo generar pruebas ejecutables ni los resultados de su aplicación

2.3 Scenario-based validation and test of software –SCENT

De las propuestas analizadas esta es la que mejor describe una aplicación en la industria (Itschner *et al*., 1998). Además, describe detalladamente cómo elicitar los requisitos, pero le hace falta detallar cómo generar los casos de prueba, lo que dificulta su aplicación. Al revisar la documentación relacionada se encontraron inconsistencias, como que la descripción hecha por Ryser & Glinz (1999) no es consecuente con la de 2003, lo que dificulta su plena comprensión y aplicación. Introduce el concepto y la notación de los diagramas de dependencias, aunque no tienen relevancia para generar casos de prueba desde casos de uso individuales. En la Tabla 4.4 se detallan sus fortalezas y dificultades.

TABLA 4.4
Fortalezas y dificultades de *SCENT*

Fortalezas	Dificultades
▪ Ofrecer un método extenso y detallado para elicitar y describir los escenarios ▪ Existir documentación de su aplicación en proyectos reales ▪ Verificar el comportamiento de un caso de uso con base en las dependencias ▪ Aplicarse en las etapas iniciales del ciclo de vida	▪ Falta de rigurosidad en la generación de los casos de prueba ▪ Muchos de los pasos los define como informales y no tiene un guía clara de cómo realizarlos, por lo que no es posible su automatización completa ▪ No garantizar que las pruebas tengan cobertura total ▪ No tener referencias a herramientas que la soporten

▪ Detallar los problemas frecuentes que aparecen en la industria del desarrollo de software y la posibilidad de adaptarla para buscarles solución	▪ No detallar cómo reducir el conjunto de casos de prueba ▪ No existir uniformidad en el contenido de su documentación

2.4 A UML-Based approach to system testing –TOTEM

Aunque es adaptable y se lograron resultados al aplicar el proceso descrito, y el ejemplo ofrecido es bastante completo, no tiene una plantilla para documentar los requisitos funcionales, es demasiado extensa y su documentación es escasa. La falta de automatización es un problema en la fase de generación de secuencias de casos de uso ya que se pueden obtener muchas combinaciones. Es una propuesta aún sin terminar pero tiene varios trabajos que le dan continuidad, por lo que se espera que en el futuro supere estos problemas. Hace falta alguna referencia a trabajos prácticos para tenerlos como guía. La Tabla 4.5 contiene sus fortalezas y dificultades.

TABLA 4.5
Fortalezas y dificultades de *TOTEM*

Fortalezas	Dificultades
▪ Describir cada paso que cubre la documentación de forma amplia y precisa ▪ Ofrecer la alternativa de generación de casos de prueba desde la descripción o las secuencias de casos de uso ▪ Incluir un ejemplo detallado y amplio	▪ No describir todos los pasos por lo que da la impresión de estar incompleta ▪ Dejar muchas de las decisiones en manos de los ingenieros de pruebas ▪ Al obtener pruebas desde diagramas de secuencias es posible conocer detalles internos e innecesarios del sistema ▪ No detallar cómo reducir el tamaño de la prueba ▪ No referenciar herramientas que la soporten ▪ La intervención de los ingenieros impide su automatización

2.5 Generating test cases from use cases

Esta propuesta no sistematiza el proceso para generar los casos de prueba, y deja a criterio del probador la aplicación de cada uno de sus tres pasos, por lo que fue posible obtener los mismos resultados al aplicar la intuición en vez de la propuesta. La Tabla 4.6 detalla sus fortalezas y dificultades.

TABLA 4.6
Fortalezas y dificultades de *Generating test cases from use cases*

Fortalezas	Dificultades
▪ Ser sencilla, fácil de comprender y ágil para aplicar ▪ Ofrecer cobertura completa de cada caso de uso por tener en cuenta todas las combinaciones posibles	▪ Tratar los casos de uso de forma aislada ▪ Dificultad para encontrar todas las combinaciones en casos de uso complicados ▪ Elevar el número de casos de prueba cuando el caso de uso es complejo ▪ No explicar cómo se aplica desde los primeros momentos del proceso del sistema ▪ No poseer alguna regla de posible sistematización ▪ No mencionar herramientas de soporte ▪ No describir casos reales de aplicación

2.6 Testing from use cases using path analysis technique

Una de sus características más importantes es que describe de forma sencilla cómo seleccionar los caminos de acuerdo con su importancia, y su dificultad es el hecho de que es difícil aplicarla en sistemas complejos. No utiliza los modelos existentes para describir requisitos y, aunque no los define, detalla los componentes que deben tener: nombre, descripción, actores o roles, precondiciones, camino básico, caminos alternativos, caminos de excepción, post-condiciones, resultados esperados, y notas. En la Tabla 4.7 se describen las fortalezas y dificultades encontradas para esta propuesta.

TABLA 4.7
Fortalezas y dificultades de *Testing from use cases using path analysis technique*

Fortalezas	Dificultades
▪ Ser sencilla y fácil de aplicar y ejecutar ▪ Ofrecer una guía para eliminar caminos sin importancia ▪ Tener un ejemplo aplicativo ▪ Describir detalladamente el proceso	▪ No incluir referencias a aplicaciones reales ▪ No indicar cómo implementar los casos de pruebas ▪ No tener en cuenta las dependencias de los casos de uso ▪ No incluir casos de uso con bucles ▪ No detallar cómo obtener los valores de prueba ▪ No detallar la implementación de los casos de prueba generados ▪ Exigir gran conocimiento del sistema para poder aplicar el proceso ▪ No detallar adecuadamente el proceso de aplicación

2.7 Use case derived test cases

Aunque es muy fácil comprender sus pasos y aplicarla, trata algunos temas superficialmente y le falta documentación. Es interesante su grado de experimentación en los procesos reales de una empresa, pero le falta más trabajo académico, ya que sólo se encontraron tres referencias. Tiene limitaciones al usar casos de uso para generar casos de prueba, ya que no utiliza los casos de uso para modelar la capacidad y el rendimiento de los requisitos relacionados, sino para modelar los requisitos funcionales, por lo que éstos se deben verificar por fuera de los casos de prueba generados, aunque la propuesta no especifica cómo realizar esa verificación.

Dado el alto número de casos de uso resultantes y el aún mayor de escenarios, fue necesario priorizar de alguna manera la generación de los casos de prueba para poder obtener un conjunto de aplicación razonable, pero la documentación tampoco detalla cómo hacerlo. Además de priorizar las pruebas, los casos de uso también pueden verificar pruebas de unidad, de integración, y el tiempo utilizado, extendiendo así el esfuerzo de la prueba a lo largo del ciclo de vida del proyecto, lo que fue fácil deducir en la aplicación de la propuesta. El modelo para diseñar los escenarios es completo e ilustra gráficamente los caminos que los datos toman en determinadas circunstancias, con lo que se determinaron los cambios de estado en los objetos. El resumen de sus fortalezas y dificultades se describe en la Tabla 4.8.

TABLA 4.8
Ventajas y dificultades de *Use case derived test cases*

Fortalezas	Dificultades
▪ Los casos de prueba obtenidos se pueden utilizar en varios tipos de pruebas ▪ El modelo de escenarios es claro y fácil de interpretar	▪ Falta claridad en la definición de qué es especificación de requisitos del software ▪ Dada la gran cantidad de casos de prueba y escenarios que resultan se dificulta su aplicación en sistemas grandes ▪ No detalla cómo probar los requisitos no funcionales ▪ Parece ser un trabajo muy experimental y tiene poca investigación académica

2.8 Requirements by Contracts allow Automated System Testing

Sus puntos fuertes son: tener diferentes criterios de cobertura para recorrer el modelo de ejecución; contar con una herramienta preliminar que la soporte –UCTSystem–; y permitir generar casos de prueba que involucran secuencias de casos de uso. Pero su dificultad es no permitir el diseño de casos de prueba para verificar aisladamente el comportamiento de cada caso de uso.

A partir del análisis a la aplicación se concluye que esta propuesta aún no está completa, ya que al partir de la especificación funcional no fue posible generar un conjunto de casos de prueba suficiente. Además, el estudio que realiza al problema es incompleto, ya que no tiene en cuenta

elementos que pueden considerarse fundamentales, como generar valores concretos de prueba o evaluar la cobertura de los requisitos. Le hace falta consistencia debido a que describe que es posible obtener directamente el conjunto de casos de prueba –valores de prueba, interacciones con el sistema y resultados esperados–, pero en la práctica no fue posible obtener casos de prueba directamente ejecutables ni generar los resultados esperados, a cambio se obtuvo un conjunto de tablas en formato propio.

No incluye cómo evaluar la calidad del conjunto de casos de prueba generado; parte del concepto de que es suficiente con seguir adecuadamente sus pasos para obtener un conjunto de casos de prueba de calidad y de cobertura máxima. Además, la automatización del proceso de generación del conjunto de casos de prueba no fue completamente posible con la herramienta, ya que los requisitos los describe en lenguaje natural. El listado de fortalezas y dificultades de esta propuesta se resumen en la Tabla 4.9.

TABLA 4.9
Fortalezas y dificultades de *Requirements by Contracts allow Automated System Testing*

Fortalezas	Dificultades
▪ Contar con una herramienta de libre distribución ▪ Generar pruebas que se basan en las secuencias de los casos de uso ▪ Utilizar contratos para expresar dependencias de los casos de uso	▪ Extender los casos de uso de forma no estandarizada por UML ▪ No permitir que se desarrollen pruebas para verificar aisladamente el comportamiento de cada caso de uso ▪ No detallar con qué criterio se selecciona el número de parámetros para un caso de uso ▪ No hay forma de relacionar los parámetros simbólicos con los valores reales ▪ Muchos de los conceptos que utiliza no tienen una definición o descripción suficiente ▪ No tiene una descripción detallada de cómo implementar las pruebas

2.9 Use case-based testing of product lines

A pesar de que sus autores aseguran que se diseñó para familias de productos, se comprobó que es más una demostración de cómo aplicar una propuesta de los años 80 a los sistemas modernos. Al modernizar la propuesta de Ostrand & Balcer (1998) también permite su aplicación a casos de uso que no pertenecen a familias de productos. Tiene como principal fortaleza ofrecer una forma de generar conjuntos de valores de prueba, sin embargo, muchos pasos los define con ambigüedad y los probadores deben cumplir demasiadas funciones de acuerdo con su experiencia. El listado de fortalezas y dificultades se describe en la Tabla 4.10.

TABLA 4.10
Fortalezas y dificultades de *Use case-based testing of product lines*

Fortalezas	Dificultades
▪ Ofrecer un ejemplo práctico completo ▪ Permitir la verificación del comportamiento de un caso de uso de acuerdo con sus dependencias ▪ Posibilitar su aplicación desde etapas tempranas del ciclo de vida ▪ Indicar cómo reducir el número de casos de prueba	▪ Dejar a discreción de los probadores muchas de las elecciones ▪ No ofrecer adecuada información sobre la cobertura de la prueba ▪ No tener referencias a herramientas de soporte ▪ No permitir la automatización total ▪ Ser inútil en los casos de uso que no manipulan datos ▪ No clarificar cómo estimar el número de casos de prueba necesarios para verificar la implementación de un caso de uso ▪ No indicar cómo identificar categorías y elecciones ▪ No explicar cómo mezclar y agrupar las pruebas en *scripts*

3. CONCLUSIONES Y RESULTADO GENERAL DEL ANÁLISIS

En la Tabla 4.11 se presenta el resumen de los resultados del análisis y el valor de los indicadores luego de la aplicación de las propuestas al estudio de caso para los criterios de evaluación.

TABLA 4.11
Análisis comparativo y valor cualitativo de los indicadores de evaluación

Propuesta **Indicador**	***Automated test case generation from dynamic models***	***Extended use case test design pattern***	***Scenario-based validation and test of software***	***A UML-based approach to system testing***	***Generating test cases from use cases***	***Testing from use cases using path analysis technique***	***Use case derived test cases***	***Requirements by contracts allow automated system testing***	***Use case-based testing of product lines***
Casos prácticos	No	No	Sí	No	No	No	No	No	No
Cobertura	Varios	Análisis de caminos	Análisis de caminos	Varios	Análisis de caminos	Análisis de caminos	Análisis de caminos	Varios	Partición de categorías
Dependencias	Sí	No	Sí	Sí	No	No	No	Sí	Sí
Documentación	Media	Media	Media	Alta -Incompleta-	Media	Media	Baja	Media	Baja
Ejemplos	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí	Sí
Estándares	Sí	No	No	Sí	No	No	Sí	Sí	No
Fecha	2000	2000	2000	2001	2002	2002	2002	2003-2004	2004
Herramientas	Sí -STRIP	No	No	No	No	No	No	Sí	Sí
Modelo	Sí	No	Sí	Sí	No	Sí	No	Sí	No
Momento de detención	No	Sí	No	No	No	No	No	No	No
Momento de inicio	Elicitación requisitos	Elicitación requisitos	Elicitación requisitos	Análisis del sistema	Elicitación requisitos	Elicitación requisitos	Elicitación requisitos	Elicitación requisitos	Elicitación requisitos
Notación gráfica	Sí	No	Sí	Sí	No	Sí	No	Sí	No
Nueva notación	No	No	Sí	Sí	No	Sí	No	Sí	No
Optimización	No	IEEE	No	No	No	Sí	No	Sí	Sí
Orden	No	No	No	No	No	No	No	No	No
Origen	Casos de Uso	Casos de Uso	Necesidades	Varios	Casos de Uso	Casos de Uso	Casos de Uso	Casos de Uso	Casos de Uso
Pasos	7	4	18	8	3	5	3	4	8
Prioridad de Casos de Uso	Sí	No	No	Sí	No	Sí	No	No	Sí
Propuesta activa	Alta	Baja	Media	Alta-Alta	Alta	Media	Media	Media	Media
Resultados	Secuencias de transiciones	Casos de prueba en lenguaje no formal	Casos de prueba en lenguaje no formal	Secuencias de transiciones	Casos de prueba en lenguaje no formal	Casos de prueba en lenguaje no formal	Casos de prueba en lenguaje no formal	Secuencias de transiciones	Casos de prueba en lenguaje no formal
Valores de prueba	No	Sí	No	No	No	No	No	No	Sí

También se seleccionó una escala numérica entre 1 y 5 para cada indicador con el objetivo de determinar la métrica para el análisis de las propuestas, con la que se valoró y determinó el posicionamiento de cada una en una métrica valorativa. El dominio se determinó de acuerdo con los resultados del estudio de caso en la aplicación de cada una de las propuestas. Los resultados obtenidos de acuerdo con esta métrica se presentan en la Tabla 4.12.

TABLA 4.12
Resultados de la métrica de valoración cuantitativa a los indicadores

Propuesta / **Indicador**	***Requirements by contracts allow automated system testing***	***Automated test case generation from dynamic models***	***A UML-based approach to system testing***	***Testing from use cases using path analysis technique***	***Use case-based testing of product lines***	***Extended use cases test design pattern***	***Scenario-based validation and test of software***	***Use case derived test cases***	***Generating test cases from use cases***
Casos prácticos	2	2	2	2	2	2	4	2	2
Cobertura	5	5	5	4	4	4	4	4	4
Dependencias	3	3	3	2	3	2	3	2	2
Documentación	3	3	4	3	2	3	3	2	3
Ejemplos	4	4	4	4	4	4	4	4	4
Estándares	5	5	5	2	2	2	2	5	2
Fecha	--	--	--	--	--	--	--	--	--
Herramientas	5	5	1	1	5	1	1	1	1
Modelo	5	5	5	5	1	1	5	1	1
Momento Detención	1	1	1	1	1	5	1	1	1
Momento Inicio	--	--	--	--	--	--	--	--	--
Notación gráfica	5	5	5	5	1	1	5	1	1
Nueva notación	4	2	4	4	2	2	4	2	2
Optimización	5	1	1	5	5	5	1	1	1
Orden	--	--	--	--	--	--	--	--	--
Origen	5	5	4	5	5	5	3	5	5
Pasos	--	--	--	--	--	--	--	--	--
Prioridad	2	3	3	3	3	2	2	2	2
Propuesta activa	3	4	5	3	3	2	3	3	4
Resultados	3	3	3	2	2	2	2	2	2
Valores de prueba	1	1	1	1	5	5	1	1	1
Valoración total	**61**	**57**	**56**	**52**	**50**	**48**	**48**	**39**	**38**

En la Tabla 4.13 se presenta la valoración general luego de analizar los resultados de las Tablas 4.11 y 4.12 de la aplicación de las propuestas al estudio de caso.

TABLA 4.13
Valoración general de las propuestas

Concepto	**Propuesta**
Propuesta más fácil de aplicar	*Generating test cases from use cases*
Propuesta más difícil de aplicar	*Automated test case generation from dynamic models*
Propuesta más ágil de aplicar	*Generating test cases from use cases* *Testing from use cases using path analysis technique*
Propuesta más pesada de aplicar	*A UML-Based approach to system testing* *Use case-based testing of product lines*
Propuesta con casos de prueba más detallados	*A UML-Based approach to system testing*
Propuesta con casos de prueba menos detallados	*Automated test case generation from dynamic models* *Use case derived test cases*
Propuesta con menos detalle	*Scenario-based validation and test of software* *Use case derived test cases, Generating test cases from use cases*
Propuesta con más detalle	*A UML-based approach to system testing*
Propuesta que no pudo aplicarse totalmente	*Generating test cases from use cases* *Use case-based testing of product lines*

Las conclusiones de los análisis anteriores se detallan en la Tabla 4.14. Estas conclusiones constituyen la base sobre la que se estructura una nueva propuesta para diseñar casos de prueba desde los casos de uso que recoge las fortalezas de las propuestas analizadas y presenta alternativas de solución a sus dificultades.

TABLA 4.14
Conclusiones finales al análisis y valoración de las propuestas

1. Ninguna propuesta se puede considerar completa
2. No hacen un estudio completo al problema
3. La documentación es escasa y deficiente
4. No tienen en cuenta trabajos previos y la mayoría son trabajos aislados
5. Falta consistencia al presentar resultados
6. No evalúan la eficiencia y la calidad de los casos de prueba
7. Son autónomas y les falta aspectos de estandarización y automatización
8. No cubren las estrategias para generar casos de prueba
9. Existe poca referenciación a aplicaciones prácticas

1. *Ninguna de las propuestas analizadas se puede considerar completa*. No se encontró una propuesta que ofrezca todo lo necesario para que a partir de la especificación funcional en los casos de uso se pueda generar, aplicar y analizar el conjunto de casos de prueba. A la mayoría se le encontraron fortalezas durante la descripción y la aplicación, sin embargo, igualmente todas presentan dificultades para ponerlas en práctica. Esas dificultades están descritas en la Tabla 4.11 con valoración "NO", además, quedaron detalladas en la respectiva Tabla de fortalezas y dificultades en el análisis de resultados de cada una, y en la Tabla 4.12 tienen una métrica de 3 o inferior.

   Una propuesta se considera completa cuando, siguiendo los pasos detallados en su documentación, se puede aplicar en problemas reales, de tal forma que sea posible diseñar, estructurar y aplicar los casos de prueba y analizar los resultados (Kaner, 2003). De acuerdo con la métrica de la Tabla 4.12, *Requirements by contracts allow automated system testing, Automated test case generation from dynamic models* y *A UML-based approach to system testing* resultaron ser las más completas, mientras que *Generating test cases from use cases* la menos valorada en cuanto a la completitud.

2. *No hacen un estudio completo al problema*. Kaner (2003) considera que un estudio del problema es completo cuando cubre todos los aspectos necesarios para su comprensión, detalle, estructuración y modelado: descripción de los requisitos, cobertura, valores de prueba, descripción de escenarios, cálculo y recorrido de caminos, valores esperados de salida, y otros que corresponden a situaciones específicas de la prueba. Un estudio completo al problema debe tener cuenta la estructura de la especificación, los criterios para seleccionar valores de prueba, cómo evaluar el grado de cobertura, cómo seleccionar el subconjunto de casos de prueba, entre otras (Myers, 2004); en este caso, las propuestas no consideran la mayoría de estos aspectos.

   Conceptos como *generar valores de prueba concretos* o *evaluar el grado de cobertura de los requisitos* no son contemplados por las propuestas, aunque son elementos básicos para concretar los casos de prueba. Por ejemplo, es posible citar la falta de información acerca de cómo generar valores de prueba en *Test cases from use cases* y *Testing from use cases using path analysis technique*, o la carencia de criterios para generar casos de prueba para casos de uso no secuenciales en *Generating requirements by contracts*; mientras que *Requirements by contracts allow automated system testing,* considerada la más completa en este análisis, aunque todavía incompleta de acuerdo con sus propios autores, tampoco describe detalladamente todos los pasos para generar el conjunto de casos de prueba.

3. *La documentación es escasa y deficiente*. Aunque en su mayoría estas propuestas son artículos en los que el detalle y la profundidad no son exhaustivos, cuando logren seguimiento y aparezcan posteriores trabajos que les den continuidad, podrán mejorarse y complementarse mediante una página web o un informe técnico que logre este objetivo. De la claridad y completitud con que se documente una propuesta depende su comprensión, difusión, aplicación y aceptación por parte de la comunidad (Gutierrez *et al*., 2006), esto se vivencia al aplicarla cuando aparecen situaciones que no es posible solucionar con la documentación, y es necesario recurrir a la experiencia, conocimientos o intuición del probador.

   En la Tabla 4.11 se muestra que el indicador de documentación tiene una valoración general "media", una cuestión que resta eficiencia a las propuestas analizadas. Las que poseen indicador de documentación menos valorado fueron *Use case derived test cases* y *Use case-based testing of product lines*, mientras que la mejor valorada fue *A UML-based approach to system testing*.

4. *No tienen en cuenta trabajos previos y la mayoría son trabajos aislados*. Estas propuestas son el resultado de investigaciones desarrolladas aisladamente que no describen un estudio al estado del arte, no analizan investigaciones a propuestas existentes, no aprovechan los puntos fuertes de otros trabajos, ni presentan alternativas a sus dificultades de aplicación. Es el caso de un conjunto de propuestas similares como *Test cases from use cases, Extended use case test desgin pattern* y *Testing from use cases using path analysis technique* que, aunque se basan en el análisis de caminos, no aportan detalles relevantes o nuevos a ese respecto, lo que demuestra la falta de estudios para analizarse entre sí. Una excepción es *Use case-based testing of product lines* que parte de un trabajo anterior y lo complementan posteriormente con el trabajo analizado en esta tesis.

5. *Falta consistencia al presentar resultados*. A pesar de que la mayoría de propuestas manifiesta que es posible obtener el conjunto de casos de prueba luego de seguir los pasos que describen, finalmente lo que se obtiene en la aplicación es un conjunto de tablas en un formato particular para cada una de ellas, en las que están estructurados los casos de prueba.

El objetivo al aplicar una propuesta desde el conjunto de valores de prueba, de interacciones con el sistema, y de resultados esperados, es obtener un conjunto de casos de prueba ejecutables sobre el sistema; pero estas tablas particulares sólo describen los casos de prueba sin ningún proceso de aplicación. Las excepciones son *Automated test case generation from dynamic models, Requirements by contracts allow automated system testing* y *Use case-based testing of product lines*, que describen herramientas de automatización para aplicar los casos de prueba obtenidos.

6. *No evalúan la eficiencia y la calidad de los casos de prueba*. Siguiendo con Kaner (2003), un caso de prueba es eficiente cuando encuentra anomalías, todas las anomalías que encuentra son errores, maximiza el recuento de errores, minimiza el costo de aplicación, encuentra escenarios seguros para el uso del producto, determina los momentos de inicio y de detención, y otras características que cada escenario de prueba pueda determinar. El objetivo de la calidad, aunque complicado debido su multi-dimensionalidad, depende de la naturaleza del producto bajo prueba y de conceptos como confiabilidad, completitud, métricas y estándares. Una propuesta que no tenga en cuenta estándares, modelos o métricas de evaluación, debe considerarse incompleta, ya que no hay forma de contrastar los resultados que se obtienen (Gutiérrez *et al*., 2006)

   Las propuestas analizadas no incluyen o referencian algún método para evaluar conceptos como eficiencia y calidad en el conjunto de casos de prueba. Con excepción de *Extended use case test desgin pattern*, que tiene en cuenta una fase de evaluación a los casos de prueba obtenidos, las demás asumen que sólo con aplicar adecuadamente sus pasos es posible obtener casos de prueba eficientes y de calidad.

7. *Son autónomas y les falta aspectos de estandarización y automatización*. En la aplicación se pudo determinar que no es posible automatizar completamente las propuestas, ya que los requisitos, como punto de partida, están descritos en lenguaje natural y muchas de las características de aplicación las deciden los probadores. Además, dado que cada una detalla sus propios pasos y utiliza formalismos y modelos particulares, igualmente su grado de automatización es particular. Otro asunto son los estándares, que la mayoría de las propuestas no los tiene en cuenta, como el formato de IEEE para requisitos y casos de uso o el modelo de casos de uso de UML.

   En cuanto a la automatización las excepciones son *Automated test case generation from dynamic models, Requirements by contracts allow automated system testing* y *Use case-based testing of product lines* que cuentan con herramientas para facilitar este proceso. Mientras que *Automated test case generation from dynamic models, A UML-based approach to system testing* y *Use case derived test cases* recomiendan alguno de los estándares propuestos por la comunidad de las pruebas del software.

8. *No cubren las estrategias para generar casos de prueba*. La experiencia demuestra que para que la aplicación de los casos de prueba sea eficiente para encontrar errores, se deben aplicar dos estrategias claramente diferenciadas: 1) evaluar todos los posibles caminos –secuencias de interacciones– descritos en los grafos, y 2) complementar los casos de prueba con pruebas orientadas a verificar las secuencias de casos de uso (Nebut *et al*., 2004).

   Como se describe en los análisis y descripciones, sólo algunas de las propuestas contemplan ambas estrategias: *Extended use case test design pattern, Testing from use cases using path*

analysis technique, *Requirements by contracts allow automated system testing* y *Use case-based testing of product lines*. Mientras que *Requirements by contracts allow automated system testing* intenta automatizarlas pero la herramienta es incipiente y no logra el objetivo. Las demás no tienen en cuenta ninguna de las estrategias y presentan una aproximación bastante empírica para hacerlo.

9. *Existe poca referenciación a aplicaciones prácticas*. Para que una propuesta adquiera el nivel de comprensión y continuidad requerido actualmente es necesario que en su aplicación pase del concepto a la práctica, ya que esto puede considerarse como un verdadero aporte al conocimiento en el área (Gutiérrez *et al.*, 2006). Con excepción de *Scenario-based validation and test of software* que describe su aplicación en un proyecto industrial, GLASS, ninguna de las otras propuestas lo hace, y esto es fundamental ya que indica el nivel de reconocimiento y aceptación por parte de la comunidad de un determinado trabajo.

3.1 Mejores características

Las propuestas analizadas presentan una serie de buenas prácticas que se toman como base para sustentar la propuesta para generar casos de prueba desde los casos uso que en esta tesis se propone. Esas características y criterios están detallados en la Tabla 4.15.

TABLA 4.15
Características y criterios comunes de las propuestas analizadas

1. Diseñar un conjunto de casos de prueba lo más completo posible para garantizar que el sistema cumpla con la especificación funcional y asegurar su calidad
2. Partir de requisitos funcionales del sistema
3. Utilizar el análisis de caminos o de estados
4. No requerir formalización de los requisitos funcionales ya que parten de una descripción en lenguaje natural
5. No contemplar la automatización mediante herramientas software, por lo que no es posible generar casos de prueba desde los requisitos funcionales de forma automática y sistemática
6. Utilizar alguna forma conocida para representar la especificación: casos de uso, diagramas de estados o escenarios
7. Permitir verificar y comprobar la correctitud y completitud de los requisitos funcionales desde las primeras fases del ciclo de vida

El objetivo de las pruebas funcionales es verificar que los requisitos de la especificación funcional se implementen adecuadamente en el desarrollo del sistema, por lo que son el punto de partida para diseñar una propuesta que, partiendo de los casos de uso, se oriente a diseñar casos de prueba para verificar los aspectos funcionales del software. Generar un modelo del sistema desde su especificación funcional consiste en construir un modelo, formal o semi-formal, para expresar el comportamiento esperado de acuerdo con la información de la especificación de requisitos. Pero, debido a que la generación de ese modelo parte de requisitos descritos en lenguaje natural, su construcción no es automática. Las propuestas analizadas en este estudio plantean la realización de esta tarea de forma sistemática a través de un conjunto de pasos claramente definidos.

Algunas de las propuestas proponen su propio criterio de cobertura para generar los casos de prueba y lo aplican sobre el modelo del sistema. El criterio más frecuentemente utilizado es el análisis de todos los posibles caminos de ejecución, pero es posible definir y utilizar otros criterios como lo hace *Requirements by contracts allow automated system testing*. Además, utilizando una herramienta software se puede automatizar el proceso, para lo que también se requiere algunos criterios para obtener secuencias de casos de uso.

Otro elemento importante para diseñar casos de prueba es definir los valores de entrada para probar el sistema. Una propuesta para generar casos de prueba desde los casos de uso será completa cuando contemple el proceso para generar valores concretos y representativos, algo que describen y aplican deficientemente las propuestas analizadas.

Desde los caminos de ejecución y los valores de prueba es posible diseñar los escenarios de prueba –una descripción abstracta del caso de prueba–, y refinarlos sin reducir su número ni afectar su calidad, una tarea realizable mediante la generación de caminos de ejecución con criterios de cobertura que reduzcan su número. Pero, como se describió en el análisis, un número alto de valores de prueba igualmente puede incrementar el número de casos de prueba, ya que puede ejecutarse varias veces con valores de prueba diferentes. Es por esto que lo más adecuado es reducir el número de casos de prueba luego de generar los caminos y de seleccionar los valores de prueba.

Los resultados esperados luego de aplicar los casos de prueba es otro elemento importante en la prueba del software. Esta actividad tiene como objetivo determinar las respuestas que ofrecerá el sistema luego de ejecutar la prueba, y se obtienen al aplicar automáticamente el conjunto de casos de prueba a simulaciones del modelo de comportamiento.

La validación del conjunto de casos de prueba para evaluar la calidad de las pruebas generadas también es una cuestión a considerar. Por ejemplo, *Extended use case test design pattern* utiliza el principio de medir el grado de cobertura de los requisitos para evaluar su calidad y, si el nivel de calidad que obtiene no se considera adecuado, repite el proceso utilizando otra selección de criterios o refinando los escenarios de prueba. Además, para que el proceso de prueba llegue a ser completo se requieren pruebas ejecutables sobre el sistema bajo prueba. La generación de código para la prueba, aunque se menciona como un paso en algunas de las propuestas, no se aborda detalladamente. El principal inconveniente es que ninguna de las propuestas genera un conjunto de casos de prueba expresados formalmente y fácilmente traducible a código; todas generan casos de prueba descritos en lenguaje natural, a excepción de las que cuentan con herramientas aunque incompletas, para generar pruebas ejecutables: *Automated test case generation from dynamic models, Requirements by contracts allow automated system testing* y *Use case-based testing of product lines*.

REFERENCIAS

- Binder, R. V. (2000). “Testing Object-Oriented Systems”. USA: Addison-Wesley.
- Bohem, B., Brown J. R., Kaspar H., Lipow M., McLeod G. & Merritt M. (1978). “Characteristics of Software Quality”. USA: North Holland.
- Engineering”. *IEEE Internet Computing*, Vol. 2, No. 3, pp. 46-52.
- Grady, R. B. (1992). “Practical software metrics for project management and process improvement”. New York: Prentice Hall.
- Gutiérrez, J. J., Escalona, M. J., Mejías, M. & Torres, J. (2005). “Estudio comparativo de propuestas para la generación de casos de prueba a partir de requisitos funcionales”. http://www.lsi.us.es/docs/informes/LSI-2005-01.pdf. Consultado Dic. 2009.
- Gutiérrez, R. J. J., Escalona C. M. J., Mejías R. M. & Torres J. V. (2006). “Generation of test cases from functional requirements: a survey”. *Workshop on System Testing and Validation*, Vol. 4, No. 4, pp. 117-126.
- Helmbold, D. & Luckham D. (1985). “TSL: task sequencing language”. *ACM SIGAda Ada Letters*, Vol. V, No. 2. Special edition of Ada Letters, pp. 255-274.
- ISO (2001). ISO/IEC TR 9126-1. “Software engineering: Product quality. Part 1: Quality model”. Switzerland: ISO copyright office.
- Itschner, R., Pommerell, C., Rutishauser, M. (1998). “GLASS: Remote Monitoring of Embedded Systems in Power
- Kaner, C. (2003). “What is a good test case?” *StarEast conference 03*. Orlando, USA, pp. 34-50.
- Myers, G. J. (2004). “The Art of Software Testing”. New York: John Wiley & Sons.
- Nebut, C., Fleurey F., Le Traon Y. & Jezequel J-M. (2004). “A Requirement-Based Approach to Test Product Families”. *Lecture Notes in Computer Science*, No. 3014, pp. 198-210.
- Ostrand, T. J. & Balcer M. J. (1988). “The Category-Partition Method for specifying and generating functional tests”. *Comunications of the ACM*, Vol. 31, No. 6, pp. 676-686.
- Pressman, R. (2004). “Software Engineering: A Practitioner's Approach”. USA: McGraw-Hill Higher Education.
- *Quality Week Europe ’99*. Brussels, Belgium.
- Ryser, J. & Glinz, M. (1999). “A Practical Approach to Validating and Testing Software Systems Using Scenarios”.
- Srivastava, V. K., Farr W. & Ellis W. (1997). “Experience with the Use of Standard IEEE 982.1 on Software Programming”. *International Conference and Workshop*. Monterey, CA, USA, pp. 121-127.
- Wagner, S (2004). “Efficiency Analysis of Defect-Detection Techniques”. *Technical Report TUMI-0413*, Institut für Informatik, Technische Universität München.
- Weitzenfeld, A. (2004). “Object Oriented Software Engineering with UML, Java and Internet”. USA: Thompson.

V. PROPUESTA Y TRABAJO FUTURO

1. PLANTEAMIENTO DE UNA NUEVA PROPUESTA

Luego de aplicar, comparar y analizar las propuestas seleccionadas, con los resultados se estructura una nueva propuesta para generar casos de prueba desde los casos de uso para verificar los aspectos funcionales del software. A continuación se describe el proceso para esa estructuración, cuyo objetivo es recoger los trabajos analizados para potencializar sus mejores prácticas y plantear soluciones a sus dificultades. En la Fig. 5.1 se representa el diagrama de actividades que resume el proceso ideal para la esta estructuración, en el que se definen los pasos a seguir a la vez que se juzga e identifican preguntas de investigación que generarán trabajos futuros para fortalecer lo que aquí se propone.

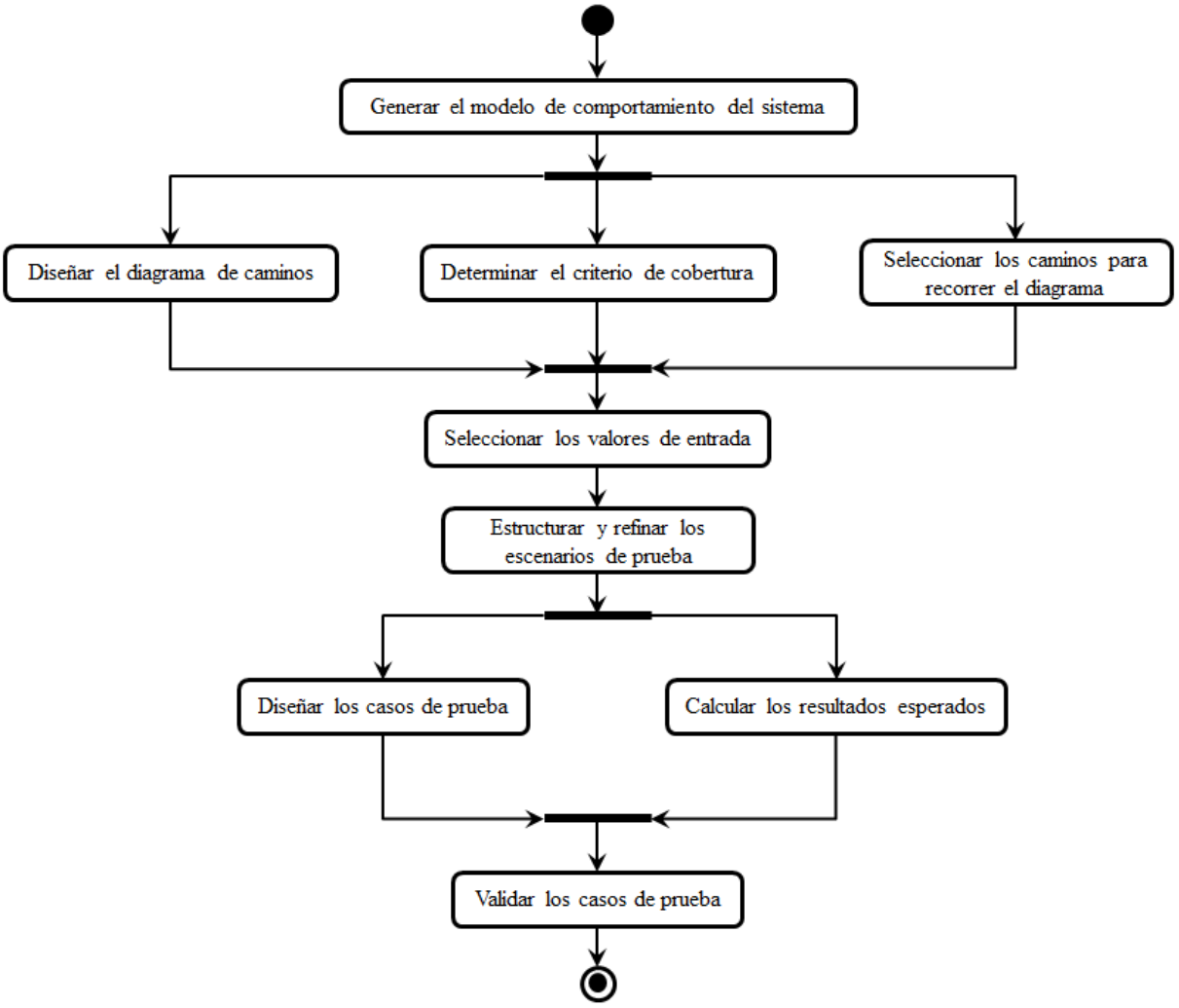

Fig. 5.1 Secuencia de actividades para estructurar la nueva propuesta

Esta nueva propuesta parte del aprovechamiento a las características comunes de las propuestas encontradas en el análisis que sustentan la base de su estructura, lo que se convierte en una ventaja ya que se utilizan las ventajas de los trabajos previos y no se repiten los procesos en los que se detectaron dificultades. Además de incluir y desarrollar todas las actividades descritas en la Fig. 5.1, se documenta cada una de forma detallada aplicando un conjunto de reglas claras y precisas para alcanzar el objetivo de lograr una propuesta de aplicación sistemática. Cada una

estas actividades presenta una serie de retos y cuestiones que no están adecuadamente resueltas en las propuestas analizadas, por lo que se asume como parte del trabajo para la propuesta que se describe.

La Tabla 5.1 detalla las actividades de la Fig. 5.1, y otras generales, y relaciona las propuestas analizadas que incluyen alguna información de cómo llevarlas a cabo, aunque las mencionen sutilmente o las propongan como trabajos futuros.

TABLA 5.1
Actividades y propuestas relacionadas

Actividad	**Propuestas**
Generar el modelo de comportamiento del sistema	*Automated test case generation from dynamic models* *Testing From Use Cases Using Path Analysis Technique* *SCENario-Based Validation and Test of Software* *A UML-based approach to system testing*
Diseñar el diagrama de caminos	Ninguna
Determinar el criterio de cobertura	*Automated test case generation from dynamic models* *Requirements by contracts allow automated system* *Testing From Use Cases Using Path Analysis Technique* *Test cases form Use Cases* *SCENario-Based Validation and Test of Software*
Seleccionar los caminos para recorrer el diagrama	Ninguna
Seleccionar los valores de entrada	*Use case-based testing of product lines* *Testing From Use Cases Using Path Analysis Technique* *Extended Use Case Test Desgin Pattern*
Estructurar los escenarios de prueba	Todas
Refinar los escenarios	*Extended Use Case Test Desgin Pattern* *A UML-based approach to system testing*
Diseñar los casos de prueba	Todas
Calcular los resultados esperados	*A UML-based approach to system testing*
Validar el conjunto de casos de prueba	*Extended Use Case Test Desgin Pattern*
Generar el código de prueba	*Requirements by contracts allow automated system*

1.1 Generar el modelo de comportamiento del sistema

Para generar este modelo y a partir de él culminar con el diseño del conjunto de casos de prueba, se deben resolver dos cuestiones: ¿Qué información debe proporcionar la especificación funcional del sistema? y ¿Cómo se debe representar? Para responderlas, se parte del análisis a la información que requieren las propuestas del estudio y de su utilidad en el proceso de generación de casos de prueba. Además, se analiza la estructura de la plantilla que describe los requisitos funcionales, y de acuerdo con esta información se evalúa la posibilidad de utilizar una plantilla existente o de proponer una nueva, de tal forma que responda a los requisitos para generar el modelo de comportamiento ideal. Otro aspecto importante, que ninguna de las propuestas analizadas tiene en cuenta, es el nivel de detalle y presentación de los casos de uso, ya que en general no son fijos e inmutables: "A medida que se avanza en el ciclo de vida del sistema los casos de uso se refinan en varias ocasiones" (Dustin *et al.*, 2002), y/o aparecen nuevos con mayor detalle, por lo que se debe determinar el nivel de detalle más adecuado con el objetivo de generar un mejor conjunto de casos de prueba.

Aunque todas las propuestas tienen como punto de partida los requisitos funcionales descritos en los casos de uso, algunas mencionan otros, por lo que la nueva propuesta tiene en cuenta otro tipo de requisitos como los de almacenamiento, de cambios de estado, las reglas de negocio y los de diseño de bases de datos, que ayudarán para que la actividad de *seleccionar valores de entrada* tenga información de base lo más cercana posible a la esperada. También se evalúa si es suficiente, en cuanto lo que se quiere verificar, con un único modelo del sistema, si se requieren varios sub-modelos –las propuestas analizadas trabajan con un único modelo desde el que

generan las secuencias de casos de uso–, o si es necesario diseñar un modelo para cada caso de uso con el objetivo de concebir posibles interacciones a partir de ellos, y resumir los resultados en un modelo general.

El concepto de caso de uso, en la literatura y en el estado del arte, no tiene una definición uniforme. La mayoría de trabajos acogen la descripción de Jacobson (1992), quien lo define como "una serie de interacciones de un actor con el sistema" –Fig. 5.2–, pero cuando la aplican en su propuesta no van más allá de presentar, mediante una representación geométrica, a un actor que lleva a cabo una tarea en el sistema. Aunque, esta información es importante ya que los casos de prueba escenifican las interacciones de los actores con el sistema, esa forma de aplicarlo se convierte en una "caja negra" que no es adecuadamente comprensible en esta fase del ciclo de vida. En esta fase participan usuarios, analistas y clientes, por lo que se requiere mucha información en el proceso de elicitación de requisitos, y que ese diagrama no proporciona ni está contenida en la plantilla de documentación, convirtiéndose en un modelo estático del sistema que no representa su comportamiento ideal.

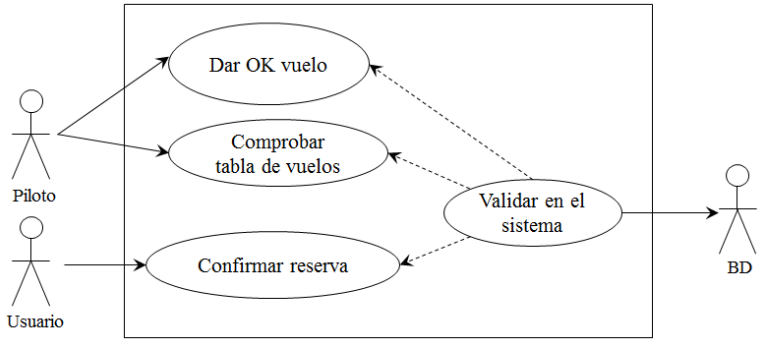

Fig. 5.2 Casos de uso sistema de vuelo (Jacobson, 1992)

Esta propuesta parte de que un caso de uso *representa una interfaz sobre la que el actor interactúa con el sistema; es un proceso que éste realiza en el sistema mediante una serie de interacciones, conformadas igualmente por una serie acciones*. Además, esa relación actor-sistema debe generar una serie de acciones-respuesta del sistema a las solicitudes de las acciones del actor. Este proceso de comunicación genera cambio de estado en las variables, los valores, las bases de datos y el hardware, que también se documentan. En la plantilla de documentación se relacionan todos los datos que no se observan en el diagrama, para de esta manera comprender la mencionada caja negra e igualmente estructurar el proceso de pruebas al sistema.

La interacción actor-sistema determina un camino principal, y posiblemente otros alternos, que un caso de prueba recorre sobre un escenario específico. Para describir esa interacción y lograr una adecuada documentación, se debe determinar un único modelo ideal de comportamiento del sistema sobre el que se ejecuten las demás actividades propuestas en la Fig. 5.1, sobre todo la de seleccionar los caminos para recorrer el diagrama y la de diseñar los casos de prueba. Además, se propone una nueva plantilla para documentar los casos de uso –Tabla 5.2–, en la que se incluye la información necesaria para generar el modelo ideal: *acción actor*, para detallar la acción que el actor realiza como parte de una interacción; *reglas del negocio*, para verificar las operaciones, si los valores resultantes son los esperados y que los casos de prueba no violen las reglas establecidas; *acción/respuesta sistema*, para describir las acciones-respuesta que el sistema ejecuta, sea como procesos internas o como respuesta a una acción previa del actor; *resultado de la acción*, para verificar que el cambio de estado en la BD y en las variables operacionales sea el

esperado. Toda esta información se expresa lo más formal que sea posible con el objetivo de lograr la representación de la Fig. 5.3, y para alcanzar la automatización del proceso de pruebas. *Esta última actividad no se aborda en esta tesis y se deja como trabajo futuro para un nuevo proyecto de investigación.*

TABLA 5.2
Plantilla propuesta para documentar casos de uso

Nombre Caso de Uso				
Código Caso de Uso				
Descripción Caso de Uso				
Nombre Interacción				
Código Interacción				
Descripción Interacción				
Actor(es)				
Pre-condiciones				
Dependencias anteriores				
Camino Principal	**Acción actor**	**Reglas del negocio**	**Acción/respuesta sistema**	**Resultado de la acción**
Caminos Alternativos				
Dependencias posteriores				
Post-condiciones				
Autor				
Fecha modificaciones				
Observaciones				

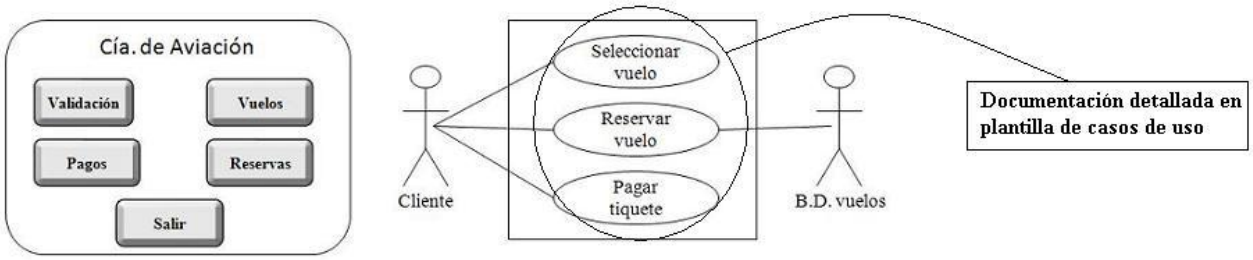

Fig. 5.3 Interpretación de casos de uso para la nueva propuesta

Con esta información se diseñan casos de prueba para verificar especificaciones como:

1. Los actores llevan a cabo las interacciones descritas y obtienen los resultados esperados.
2. Los caminos del proceso principal y alternos descritos en el caso de uso ocurren de acuerdo con las condiciones documentadas, y toman el camino apropiado en cada circunstancia.
3. Se cumplen las reglas del negocio.
4. Los valores calculados siempre son correctos.
5. El cambio de estado en la BD y en las variables operacionales ocurren correctamente.
6. Los casos de uso dependientes tienen un comportamiento adecuado.

Además, con el objetivo de estandarizar procesos y documentación, esta propuesta define un caso de uso útil, para lo que parte de la definición de Jacobson (1992): "está constituido por una serie de interacciones de un actor con el sistema", pero a la que agrega, de acuerdo con Arango y Zapata (2006), que ese conjunto de interacciones "debe tener un propósito bien definido: permitir que el actor ejecute una serie de acciones sobre el sistema, de tal forma que logre el objetivo de validar el funcionamiento". También se define interacción, un término de por sí ambiguo, como "un conjunto de acciones que lleva a cabo una tarea –consulta o transacción– representada en el diagrama de procesos" (Arango & Zapata, 2006). Entonces, un caso de uso será un "fragmento" del diagrama de procesos con un propósito definido, que de hecho también es ambiguo, y en consecuencia una decisión del modelador, pero que será más simple si lo hace sobre el diagrama de procesos, como se observa en la Fig. 5.4.

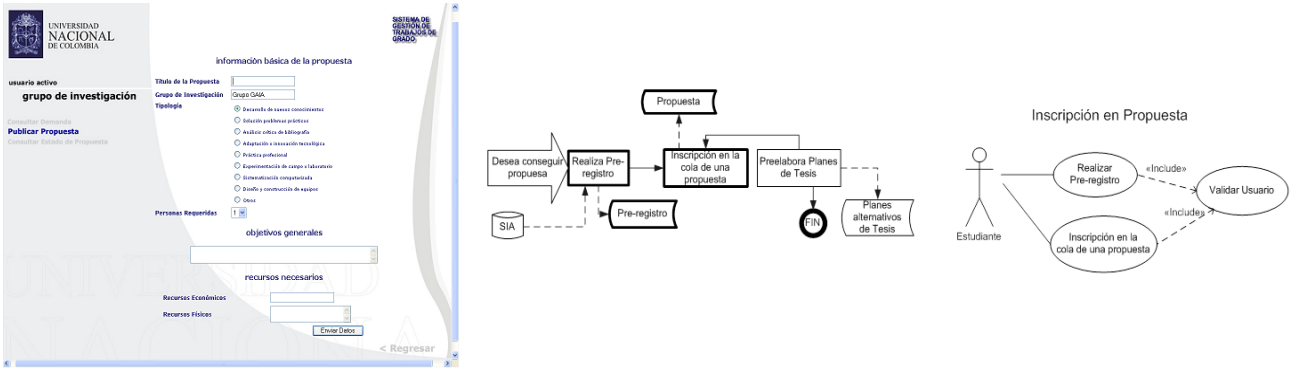

Fig. 5.4 Proceso de las interacciones del actor con el sistema (Arango & Zapata, 2006)

Para presentar las reglas del negocio se toma la propuesta de Arango & Zapata (2004) de alinear las metas organizacionales con la elicitación de requisitos mediante el uso de diagramas causa-efecto. Esta propuesta consiste en utilizar el diagrama causa-efecto para registrar las ligaduras que existen entre las metas y reglas de la organización con las características de una posible solución. Para ello construye una serie de diagramas causa-efecto –Fig. 5.5– que reemplazan progresivamente las metas y reglas organizacionales, asociadas con el entorno de aplicación del software, con factores o metas más simples hasta llegar finalmente a la caracterización de una serie de factores informáticos y no informáticos, considerados elementales. "La construcción de estos diagramas permite un desplazamiento progresivo desde los modelos del negocio hasta los modelos del software propiamente dichos" (Arango & Zapata, 2004).

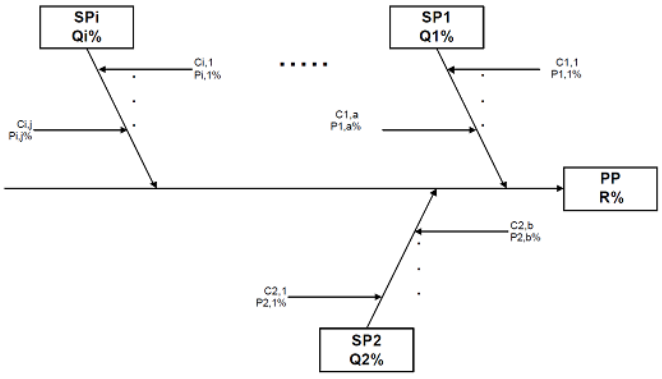

Fig. 5.5 Diagrama causa-efecto con las modificaciones planteadas en el método (Arango & Zapata, 2004)

Para describir la información que ingresa y sale de las interacciones del actor con el sistema se utiliza el formato de la plantilla propuesta en la Tabla 5.2. En ella se anota, lo más formal posible,

todos los datos que determinan las acciones ejecutadas por el actor en sus interacciones con el sistema. Es necesario entablar un diálogo continuo con el usuario para que todos los detalles de sus necesidades queden representados en el diagrama de procesos propuesto. Mediante la utilización de prototipos se valida con el usuario toda la información proveniente de la lectura a la interface, al diagrama de procesos y al diagrama causa-efecto, con lo que se obtienen los datos necesarios para registrar en la plantilla.

Los cambios de estado en la BD y en las variables operacionales se obtienen de la información registrada de las fórmulas que la solución necesita para responder a los requisitos funcionales, y para respetar las reglas del negocio. En la plantilla se registra la acción del actor, la acción-respuesta del sistema, y con estos datos se conocen los cambios de estado en la información que registra o consulta la BD, y se valida si el proceso, que origina o modifica esa acción, recorre el camino esperado en el modelo de comportamiento del sistema; además, se determinan las bases para diseñar los escenarios desde los que se estructuran los casos de prueba.

1.2 Diseñar el diagrama de caminos

Esta actividad consiste en construir un diagrama que represente el modelo ideal de comportamiento del sistema y, dada la diversidad de diagramas utilizados en las propuestas analizadas, se propone utilizar un grafo dirigido para representarlo. De la plantilla de documentación de los casos de uso se extraen las interacciones del actor, las acciones-respuesta del sistema y los cambios de estado, y se representan mediante estructuras proposicionales. Posteriormente, mediante depuraciones iterativas, se determina cuáles son las que finalmente se llevarán al grafo; para este proceso se asigna: un valor secuencial de nodo, la prioridad y dependencia de ejecución de acuerdo con el diagrama de procesos, y en las aristas del diagrama se detallan las pre y post condiciones. El grafo se convierte en una vista de la información de partida de la actividad anterior, y la principal ventaja, sobre alternativas como los diagramas de secuencias de UML o los diagramas causa-efecto que tienen múltiples interpretaciones y lecturas, es su facilidad de construcción, de lectura e interpretación. Este grafo sólo permite verificar que el ciclo de vida procede de forma adecuada y de acuerdo con lo documentado en la actividad del modelo del sistema, y que no se utiliza para verificar cuestiones como completitud o calidad de la elicitación.

El proceso de diseño del grafo consiste en leer linealmente las afirmaciones que contiene la plantilla de documentación de los casos de uso y expresarlas mediante proposiciones (Grassmann & Tremblay, 1998), como se presenta en la Tabla 5.3. Esto permitirá verificar, mediante procesos altamente formales, que las interacciones de los actores y el sistema respetan las reglas del negocio, y que en actividades posteriores se obtengan valores de entrada y valores esperados adecuados para diseñar casos de prueba. El estudio de caso es el que se utilizó para aplicar las propuestas del estado del arte, descrito en el capítulo III.

TABLA 5.3
Afirmaciones documentadas expresadas como proposiciones

El computador tiene enlace con el sistema
El nombre de usuario es alfanumérico entre 8 y 15 caracteres
El *password* es alfanumérico entre 5 y 10 caracteres
El sistema solicita al usuario el nombre de usuario y el *password*
El usuario digita el nombre de usuario
El usuario digita el *password*
El sistema valida el nombre de usuario y el *password*
El sistema verifica el nombre de usuario y el *password* en la base de datos
El sistema presenta en el cuadro de diálogo el resultado de la verificación

Debido a que el número de proposiciones resultante puede ser elevado, se aplica un proceso de depuración mediante métodos iterativos hasta obtener el listado definitivo (Saad, 2003), al que se asigna un número secuencial para convertir las proposiciones en nodos del grafo –Tabla 5.4. Además, se tiene en cuenta la prioridad y la dependencia de ejecución, ya que es posible que un nodo sea prioritario para ejecutar otro caso de uso, por lo que se convierte en dependiente del mismo. Debido a que la mayoría de propuestas analizadas no tiene en cuenta esta característica, y a que las que lo hacen utilizan un método propio, es necesario analizar adecuadamente la prioridad y dependencia de ejecución antes de diseñar el grafo de caminos, tal como se plantea, aunque en otra dirección, en Capuz *et al*. (2000).

TABLA 5.4
Proposiciones resultantes luego de las iteraciones

Prioridad	Proposiciones	Iteración
	El computador tiene enlace con el sistema	Anulada
	El nombre de usuario es alfanumérico entre 8 y 15 caracteres	Regla de negocio
	El *password* es alfanumérico entre 5 y 10 caracteres	Regla de negocio
	El sistema solicita al usuario el nombre de usuario y el *password*	Anulada
1	El usuario digita el nombre de usuario	
2	El usuario digita el *password*	
3	El sistema valida el nombre de usuario y el *password*	
4	El sistema verifica el nombre de usuario y el *password* en la base de datos	
	El sistema presenta en el cuadro de diálogo el resultado de la verificación	Anulada

Debido que el grafo dirigido representa una vista de la información de partida, es decir las interacciones de las interfaces, el diagrama de procesos, los casos de uso y su documentación, se incluyen las pre y post-condiciones documentadas, y las excepciones que hacen que los caminos del grafo tomen una ruta determinada. Este proceso se deja a criterio del probador porque cada proyecto presenta particularidades propias que no es posible estandarizar mediante un método único. En la Fig. 5.6 se propone un criterio de aplicación.

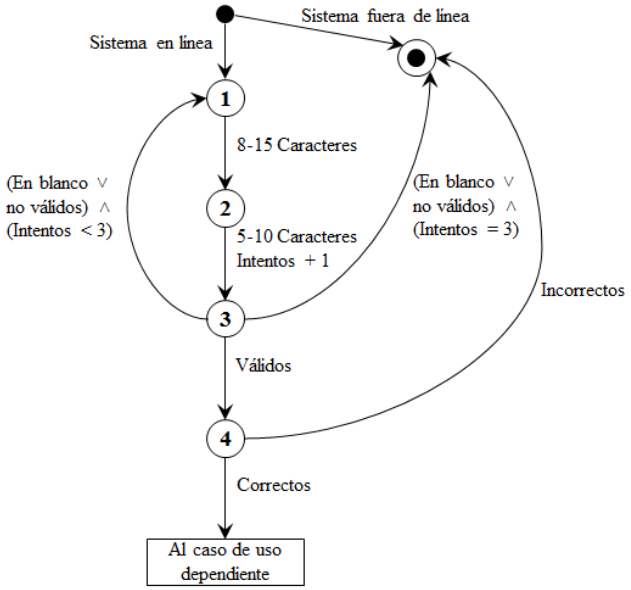

Fig. 5.6 Grafo de caminos propuesto para el estudio de caso

1.3 Determinar el criterio de cobertura y seleccionar caminos de recorrido

Para desarrollar esta actividad primero se responden cuestiones cómo: ¿Cuáles son los criterios propuestos hasta el momento y cuál es el que mejores resultados puede ofrecer? ¿Es necesario

diseñar un nuevo criterio o es posible utilizar alguno de los existentes? Las propuestas analizadas aplican criterios propios para alcanzar la cobertura y para seleccionar los caminos de la prueba. Tal es el caso de *Automated test case generation from dynamic models* que, mediante lenguaje STRIPS y un algoritmo de planificación, genera las secuencias para recorrer el modelo representado en un diagrama de estados, luego selecciona de entre dos criterios la cobertura de la prueba: 1) considerar todos los estados; y 2) considerar todas las transiciones en el diagrama de estados. Pero con estos criterios no fue posible lograr un adecuado criterio de cobertura, debido a que con el lenguaje STRIPS y el algoritmo no se obtuvieron proposiciones de fácil implementación para recorrer los caminos, por lo que cada vez fue necesario regresar y describir nuevamente las operaciones propuestas.

El criterio de cobertura propuesto por Srivastava *et al*. (1997), aplicado en *Extended use case test design pattern*, es funcional sólo para casos de uso en los que los valores de entrada son pocos, ya que al analizar todos los posibles valores que éstos pueden tener para un caso de uso extenso ese número sería demasiado elevado como para considerarlo eficiente o funcional. *Requirements by contracts allow automated system testing* propone cuatro opciones para determinar el criterio de cobertura: 1) de todas las aristas; 2) de todos los nodos; 3) de todos los casos de uso instanciados; y 4) de todos los nodos y casos de uso instanciados. Para aplicarlos utiliza la herramienta de libre distribución RBCTool[1], pero dicha herramienta no permite la ejecución del proceso en el criterio de casos de uso instanciados, ni en el de todos los nodos y casos de uso instanciados; presenta un error de bloqueo interno y detiene el proceso.

Por todo esto se concluye que las propuestas analizadas no ofrecen un método eficiente para determinar un criterio de cobertura; todas buscan caminos o identifican categorías que no son suficientes para lograr el objetivo de esta actividad. Una nueva propuesta debería unir ambas aproximaciones para recorrer los caminos del modelo del sistema, y de esa manera posiblemente se podría conocer cuántos casos de prueba diseñar cuando para recorrer un camino de ejecución con *n* valores de prueba diferentes. Asumiendo que un escenario de prueba es un camino de ejecución con un conjunto exacto de valores, se necesitarían *n* casos de prueba, pero no es tan simple ya que mucha de la calidad de la prueba depende del número de casos de prueba diseñados, y todavía no es claro cómo calcularlo.

En algunas de las propuestas analizadas y en otras investigaciones generalmente utilizan el criterio de cobertura del cartero chino –*chinese postman*– (Robinson, 2000), que consiste en recorrer todas las posibles transiciones utilizando el menor número de caminos. Pero muchas veces esto no es suficiente o el número de transiciones es tan elevado que se convierte en una empresa imposible de llevar a cabo, por lo que es necesario tener en cuenta otros posibles criterios de cobertura con el objetivo de resaltar el valor del camino principal y disminuir la importancia de los secundarios, o para resaltar los caminos que más se utilizan, como lo proponen Riebisch *et al.* (2003).

Otra característica de esta actividad es poder generar aleatoriamente los recorridos de tal manera que se obtengan varios puntos de vista para justificar cuál de ellos es el más eficiente. Todo esto se debe a que, como se observó en la aplicación de las propuestas, en la mayoría de los casos fue necesario generar dos conjuntos de casos de prueba, uno desde las interacciones de cada caso de uso y otro desde sus secuencias, por lo aquí se evalúa si es posible utilizar el mismo criterio de cobertura para ambos conjuntos o si cada uno requiere un criterio propio. Una posibilidad es

[1] RBCTool. UCTSystem. http://www.irisa.fr/triskell/results/ISSRE03/UCTSystem/UCT System.html. 08-2009.

utilizar la prioridad relativa a los casos de uso, que incorporan propuestas de ingeniería de requisitos como la de Escalona (2004) y que es necesaria para generar interacciones o secuencias entre casos de uso. Aún quedan otros interrogantes que se deben responder, como si es necesario seleccionar todos los caminos, o si es posible determinar cuántos son suficientes, pero se escapan al alcance de este trabajo y se proponen como preguntas de investigación para trabajos futuros. Desde esta tesis se propone, para calcular el número mínimo de caminos en un grafo, aplicar el método del cálculo de caminos mediante matrices binarias de la teoría de grafos en la Matemática Discreta (Grassmann & Tremblay, 1998) o el algoritmo de Dijkstra de la misma teoría.

Este método define camino como:
Sea *G = (N, A)* un grafo dirigido. Una sucesión de aristas es un *camino* de G ↔ el nodo terminal de cada arista del camino es el nodo inicial de la próxima arista del camino, si lo hubiere.

Ejemplo de caminos son: $\{(N_{i1}, N_{i2}), (N_{i2}, N_{i3}), \ldots, (N_{ik-2}, N_{ik-1}), (N_{ik-1}, N_{ik})\}$. Un camino *recorre* los nodos que aparecen en la sucesión, *comenzado* en el nodo inicial de la primera arista y *finalizando* en el nodo terminal de la última arista. El número de aristas que aparecen en la sucesión de un camino se denomina *longitud* del camino; un camino *independiente* tiene que contener al menos una arista que no esté contenida en un camino anterior; un grafo se ve entonces como un conjunto de pares ordenados *(N, A)*, donde *A* es una relación binaria en *N*; es decir, $A \subseteq N \times N$.

Para calcular los caminos en el grafo utiliza el álgebra matricial:
Sea *G = (A, N)* un grafo en el cual $N = \{V_1, V_2, \ldots, V_n\}$ con los nodos ordenados desde V_1 hasta V_n. La matriz de caminos *Ad*, *n x n*, posee elementos a_{ij} cuyo valor será:

$$a_{ij} = \begin{cases} 1, \text{ si } (v_i, v_j) \in E \\ 0, \text{ en caso contrario} \end{cases}$$

Aplicando este método al grafo de la Fig. 5.6 se obtiene *E* y luego se verifica la pertenencia de cada arista de *A* a *E*.

$$E = \begin{Bmatrix} (I,I), (F,F), (1,1), (2,2), (3,3), (4,4), \\ (I,F), (F,I), (1,I), (2,I), (3,I), (4,I), \\ (I,1), (F,1), (1,F), (2,F), (3,F), (4,F), \\ (I,2), (F,2), (1,2), (2,1), (3,1), (4,1), \\ (I,3), (F,3), (1,3), (2,3), (3,2), (4,2), \\ (I,4), (F,4), (1,4), (2,4), (3,4), (4,3), \\ (I,S), (F,S), (1,S), (2,S), (3,S), (4,S) \end{Bmatrix}$$

A = {(I,F), (I,1), (1,2), (2,3), (3,1), (3,F), (3,4), (4,F), (4,S)}

$$Ad = \begin{pmatrix} \mathbf{I} & \mathbf{F} & \mathbf{1} & \mathbf{2} & \mathbf{3} & \mathbf{4} & \mathbf{S} & \\ 0 & 1 & 1 & 0 & 0 & 0 & 0 & \mathbf{I} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{F} \\ 0 & 0 & 0 & 1 & 0 & 0 & 0 & \mathbf{1} \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & \mathbf{2} \\ 0 & 1 & 1 & 0 & 0 & 1 & 0 & \mathbf{3} \\ 0 & 1 & 0 & 0 & 0 & 0 & 1 & \mathbf{4} \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & \mathbf{S} \end{pmatrix}$$

Dado que la matriz resultante es una matriz booleana cuyos elementos son 0 ó 1 –Falso o Verdadero–, la *i-ésima* fila está determinada por las aristas que se originan en el nodo v_i, y el número de elementos cuyo valor sea 1 constituye el grado de salida de ese nodo; igualmente para

la columna *j-ésima*, el grado de entrada del nodo es la cantidad de elementos con valor 1. En el ejemplo observamos que el nodo F tiene grado de salida 0, pero grado de entrada 3; mientras que el nodo 3 tiene grado de salida 3 y de entrada 1.

Con esta información se determina la cobertura para seleccionar los caminos, es decir, obligatoriamente se debe seleccionar los caminos cuyos nodos tengan 1 como grado de salida y de entrada; los de grado 2, cuando tomen ese camino por violación a las reglas del negocio o por cambios de estado en la base de datos o las variables; y los de grado 3 o superior, sólo cuando no hayan sido recorridos previamente por los caminos de grados inferiores. Para el estudio de caso de la figura 5.6 los caminos seleccionados serán: (I, 1, 2, 3, 4, S), (I, 1, 2, 3, 1), (I, 1, 2, 3, F) y (I, 1, 2, 3, 4, F)

1.4 Seleccionar los valores de entrada

El objetivo de esta actividad es establecer un conjunto de reglas precisas y sistemáticas que permitan reducir el número de decisiones que toman los probadores cuando seleccionan los valores de entrada para recorrer los caminos. Esto se logra, como se comentó antes, incluyendo otro tipo de requisitos en la elicitación, de tal forma que se tenga acceso a mayor información para seleccionar los valores de entrada.

Sólo tres de las propuestas analizadas describen alguna forma para seleccionar los valores de entrada: *Use case-based testing of product lines*, que utiliza la plantilla y el método propuestos por Ostrand & Balcer (1988) para obtener un listado de las especificaciones de prueba –un documento que recoge todas las categorías de datos, las elecciones de valores de entrada para cada una y sus restricciones; *Testing from use cases using path analysis technique*, que propone seleccionar los valores de entrada mediante la técnica de valores de frontera o valores límite; y *Extended use case test design pattern*, que cuando define las variables operacionales describe sucintamente que la mejor forma de seleccionar los valores de entrada para esas variables es aplicar la técnica de tablas de decisión.

La técnica de valor límite propuesta por Bohem (1979) selecciona los valores de entrada en los valores de las fronteras de la variable para recorrer los caminos del grafo. Es un proceso de selección de valores de entrada –o datos de prueba– mediante el descubrimiento y comprensión de sus fronteras, diferenciando entre las condiciones válidas y no válidas de ejecución. Los caminos se recorren para verificar las fronteras internas y externas de estos valores, además de sus valores exactos.

En una partición de equivalencia (Myers, 1979) los datos de entrada de una unidad de software se dividen en una partición de datos desde la que se selecciona los valores de entrada para recorrer los caminos. El proceso consiste en tomar cada condición de entrada descrita en la especificación y calcular por lo menos dos clases de equivalencia para sus valores de entrada. Una clase representa al conjunto de valores que satisface la condición –la clase válida– y la otra representa los casos que no la satisfacen –la clase no válida. Partiendo de la información representada en la elicitación de requisitos, este método se utiliza para reducir a un conjunto finito el número total de casos de prueba.

En la técnica de tablas de decisión (Ferriday, 2006) (Vauthier, 2007) se representa las relaciones lógicas entre las condiciones de entrada y las acciones de salida, desde las que se derivan valores de entrada para diseñar casos de prueba; esto lo hace sistemáticamente considerando cada combinación de condiciones y acciones posibles. Al utilizar una tabla de decisión para identificar

valores de entrada, la propiedad de completitud de dicha tabla garantiza una prueba que recorre completamente los caminos del grafo del sistema. No hay un orden particular para las condiciones ni para las acciones seleccionadas.

En esta tesis se propone, para seleccionar los valores de entrada, aplicar el método utilizado en la propuesta *Extended use case test design pattern* con las variables operacionales mediante tablas de decisión, como se expresa en la Tabla 5.5.

TABLA 5.5
Pasos para seleccionar los valores de entrada

Paso	Actividad	Objetivo
1	Identificar las variables operacionales	Determinar la lista de variables operacionales para cada caso de uso
2	Definir el dominio de las variables operacionales	Encontrar el dominio de ejecución de cada una de las variables operacionales
3	Identificar las relaciones operacionales	Estructurar las tablas de decisión con las relaciones entre las variables operacionales

1. Una variable operacional la conforman los factores que varían entre las diferentes ejecuciones del caso de uso. Para el estudio de caso de la Fig. 5.6 las variables funcionales son: sistema en línea, nombre de usuario, y *password* de usuario.
2. El dominio de las variables operacionales está documentado en la plantilla de los casos de uso a través de las reglas del negocio. En la Tabla 5.6 se detalla el dominio resultante.

TABLA 5.6
Dominio de las variables operacionales

Variable	Domino	Tamaño
Sistema en línea	Booleano: Si/No	NA
Nombre de usuario	Alfanumérico: Serie de caracteres/En blanco	8-15
Password de usuario	Alfanumérico: Serie de caracteres/En blanco	5-10

3. Una relación operacional es aquella que incluye la combinación de valores que toman las variables operacionales, como se presenta en la Tabla 5.7.

TABLA 5.7
Relaciones operacionales resultantes

Sistema en línea	Nombre de usuario	Password de usuario
No	NA	NA
Si	En blanco	En blanco
Si	No válido	En blanco
Si	Válido	No válida
Si	Incorrecto	Incorrecta
Si	Incorrecto	Correcta
Si	Correcto	Correcta

1.5 Estructurar y refinar los escenarios de prueba

Los términos *escenario de prueba* y *caso de prueba* a menudo se utilizan como sinónimos, y las propuestas analizadas presentan diversas definiciones, aplicaciones y diseños para ellos. Por ejemplo, la plantilla para escenarios que aplica *SCENT* es bastante amplia y ofrece mucha información en la plantilla de documentación. *Testing from use cases using path analysis technique* presenta el diseño de otra plantilla que, aunque no tan completa como la anterior, también es útil en muchos casos. Una forma novedosa consiste en presentar los escenarios en forma de grafos, como lo hace *Use case derived test cases*, que combina grafos de espina de pescado con grafos dirigidos y de secuencia, y ofrece un imagen dinámica de los datos, el cambio de estado y el recorrido sobre los caminos del grafo del sistema.

Los escenarios se diseñan para representar tanto las situaciones típicas como la inusuales que pueden ocurrir en la aplicación (Davis, 2010). De acuerdo con Kaner (2003), un escenario es una historia hipotética utilizada para ayudar a una persona a comprender un problema o sistema complejo. Un escenario de prueba ideal debe tener las siguientes características:

- Es una historia acerca de cómo utilizar el programa incluyendo información de las motivaciones de las personas involucradas.
- La historia debe ser motivadora; los involucrados pueden influenciar para desarrollar pruebas bajo un escenario determinado, ya que están motivados a hacerlo debido a la posibilidad de encontrar errores en dicho escenario.
- La historia debe ser creíble; no sólo podría ocurrir en el mundo real y los involucrados deben creer que algo así probablemente puede suceder.
- La historia implica un uso complejo del programa, un entorno complejo o un conjunto complejo de datos.
- Los resultados deben ser fáciles de evaluar. Esto es valioso para todas las pruebas pero especialmente importante para los escenarios, debido a que en sí ellos son complejos.

Un buen conjunto de escenarios debe deja al probador preguntándose qué será lo más verosímil o probable; y lo obliga a pensar más y a ampliar el punto central del escenario –para aprender más acerca de futuros alternativos– con lo que tomará mejores decisiones. Por supuesto, los escenarios de este tipo son difíciles de construir, y los estudios actuales no se comprometen ampliamente con ellos. Los escenarios deben educar para la prueba, no para tratar de encontrar una solución determinada. Un escenario mal diseñado hace que al probador le sea difícil decidir cuál será el más probable o cuál logrará el mejor resultado. El mejor uso de los escenarios de prueba es para logra la comprensión del software bajo prueba, y no para tratar de predecir el futuro funcionamiento del producto.

En esta actividad se responden cuestiones como ¿Cómo enlazar los caminos de ejecución con los valores de entrada? ¿Qué estructura de escenario es la más eficiente? ¿Cuántos escenarios diseñar? Para lo que se parte de que es posible definir que un escenario A es repetido con respecto a otro B cuando se cumple que: 1) A y B se aplican con los mismos valores de prueba, lo que incluye mismas precondiciones y estados iniciales del sistema; 2) A y B ofrecen los mismos resultados esperados; y 3) las interacciones o secuencias de los casos de uso en A son iguales o están contenidas en las de B, por lo que se elimina A sin afectar la calidad de la prueba.

Las propuestas analizadas que trabajan con escenarios, *Scenario-based validation and test of software* y *Scenario-based validation and test of software,* aunque detallan mediante diagramas de estados las fases de estructuración y diseño de los mismos, no ofrecen suficiente información para refinarlos una vez están estructurados. Un procedimiento que sirve para encontrar escenarios o valores de prueba ya utilizados consiste en realizar un grafo de seguimiento al árbol de escenarios del sistema. Ese árbol es una construcción binaria de las ramas que genera el seguimiento de los caminos de prueba seleccionados, y se utiliza para detectar, mediante documentación apropiada como la que propone *SCENT* en sus diagramas de estados, cuáles escenarios se han ejecutado ya.

Algunos criterios para realizar esta labor son: utilizar una nomenclatura que identifique los caminos recorridos en cada escenario para encontrar los que se repiten más veces en los procesos de aplicación; documentar adecuadamente los escenarios de tal forma que se pueda visualizar fácilmente los que, aunque estén descritos de forma diferente, tengan mucha semejanza en su

descripción; realizar un árbol binario de seguimiento de caminos simulando la aplicación de los escenarios para encontrar las ramas recorridas varias veces por los caminos de secuencia, con lo que se detecta las que dichos caminos "visitan" varias veces.

En esta tesis se parte del hecho de que un escenario es una combinación de hipótesis, hechos y tendencias, y que es el área que necesita una mayor comprensión del sistema para presentar una solución a un problema en particular. Se llaman escenarios porque son como "escenas" en una obra de teatro: una serie de puntos de vista diferentes o presentaciones acerca de un mismo tema general. Al observar varios escenarios al mismo tiempo se comprenderán las opciones o posibilidades para diseñar los casos de prueba. El número de escenarios debe ser pequeño, pero no demasiado como para no indicar con claridad cómo recorrer los caminos seleccionados con las valores de entrada y esperar una salida determinada. Su longitud también es variable, pero pueden ser necesarias una o dos páginas: demasiado corto puede ofrecer poca información y demasiado largo puede hacer difícil su lectura y comprensión.

En la nueva propuesta se utiliza el método de *Use case derived test cases* para estructurar y refinar los escenarios, como ilustra la Fig. 5.7 del estudio de caso.

Luego se describen los escenarios mediante una tabla de seguimiento, sobre la que se refinan para luego calcular los valores esperados y culminar con el diseño de los casos de prueba. En la Tabla 5.8 se documentan los escenarios de la Fig. 5.7 sin refinar. En el estudio de caso se observa que los escenarios 3 y 4 son prácticamente iguales, y que el 5 difiere mínimamente de ellos. Al refinar estos escenarios se suprimen 3 y 5, y la acción de "número de intentos igual a 3" se agrega al escenario 4.

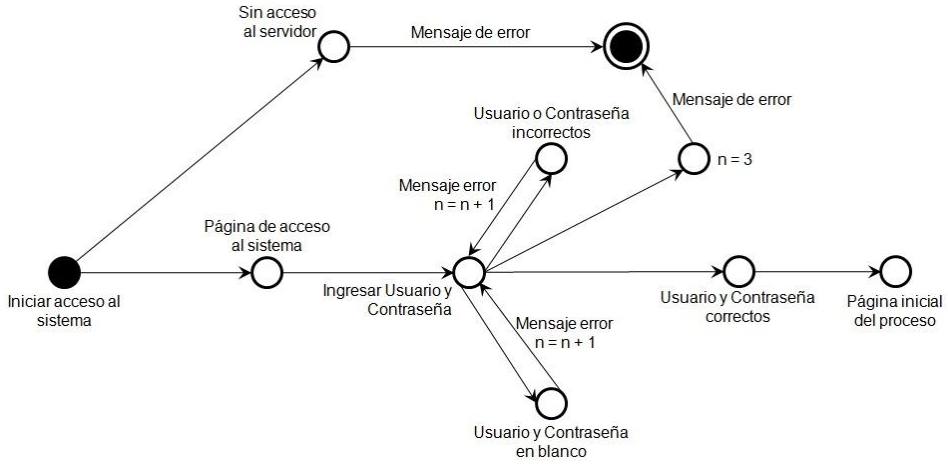

Fig. 5.7 Escenarios resultantes

TABLA 5.8
Escenarios documentados

	Escenario	**Sistema en línea**	**Nombre de usuario**	***Password* de usuario**	**Secuencia de acciones**
1	Escenario del día feliz	Si	dianatorres	diato12345	El computador tiene acceso al sistema. El sistema despliega cuadro de diálogo de acceso. El usuario digita nombre de usuario y password de usuario correctos. El sistema carga cuadro de diálogo principal.
2	Error de conexión	No			El computador no tiene acceso al sistema. Se despliega mensaje de error. Fin del proceso.

3	Nombre de usuario o password de usuario no válidos	Si	¿*diana	“ ”	El computador tiene acceso al sistema. El sistema despliega cuadro de diálogo de acceso. El usuario digita nombre de usuario o password de usuario no válidos. El sistema incrementa n en 1. El sistema despliega mensaje de error. El sistema solicita nombre de suario y password de usuario.
4	Nombre de usuario o password de usuario incorrectos	Si	dianatorres	diato1234	El computador tiene acceso al sistema. El sistema despliega cuadro de diálogo de acceso. El usuario digita nombre de usuario o password de usuario incorrecto. El sistema incrementa n en 1. El sistema despliega mensaje de error. El sistema solicita nombre de suario y password de usuario.
5	Número de intentos de ingreso al sistema igual a 3	Si	dianatorres	+++”#	El computador tiene acceso al sistema. El sistema despliega cuadro de diálogo de acceso. El usuario digita nombre de usuario o password de usuario no válidos o incorrectos. El sistema incrementa n en 1. Número de intentos de ingreso igual a 3. El sistema despliega mensaje de error. Fin del proceso.

1.6 Calcular resultados esperados y diseñar casos de prueba

La mayoría de las propuestas analizadas aplican procedimientos para especificar los valores de entrada y otras utilizan técnicas como valores de frontera o tablas de partición, pero ninguna ofrece información precisa para determinar los valores esperados o de salida antes de aplicar los casos prueba. Muchas lo dejan simplemente a criterio del probador, para que los determine con base en su experiencia y de acuerdo con el proceso previo; otras como *SCENT, Generating test cases from use cases* y *Testing from use cases using path analysis technique*, expresan en los escenarios la necesidad de contar con valores esperados pero no explican cómo calcularlos. Una alternativa, utilizada por muchos probadores, es construir *oracles* de prueba en los que ofrecen resultados válidos para cualquier valor de entrada pero este proceso no es sencillo. Construir *oracles* y generar resultados esperados para los casos de prueba es un reto de envergadura para la generación automática de pruebas (Binder, 2000), que está por fuera del alcance de este tesis y se plantea como trabajo futuro.

Para este proyecto, conocer de antemano los resultados que la aplicación de los casos de prueba ofrecerá es una actividad que se realiza de acuerdo con las reglas del negocio, las fórmulas y las excepciones elicitadas y documentadas en el modelo de comportamiento del sistema, teniendo en cuenta los valores de entrada y los escenarios estructurados en las anteriores actividades de esta propuesta. En la plantilla para documentar los casos de uso se incorporaron las reglas del negocio, en las que se especifica las fórmulas y las excepciones que los requisitos deben respetar, y con esta información se calculan los resultados esperados o por lo menos el rango en el que están ubicados. Aunque es difícil conocer el valor exacto que debe entregar un determinado caso de prueba, dadas las características específicas de cada proyecto es posible conocer un rango de valores; luego, mediante la validación de las reglas del negocio y las fórmulas especificadas en los requisitos, se determina el cumplimiento o no del caso de prueba aplicado. De acuerdo con esto, en la Tabla 5.9 se detallan los casos de prueba y los valores esperados para el estudio de caso.

Cada una de las filas de la Tabla 5.8 es un caso de prueba estructurado, cuyo conjunto se aplica al software bajo prueba para verificar el proceso. Se aclara que el estudio de caso que se utiliza para presentar cada actividad de esta propuesta es sólo un caso de uso, por lo que no se incluyen las

dependencias de casos de uso hasta el momento. En el anexo IV al final del documento se detalla la aplicación completa de las actividades de la propuesta a un estudio de caso ATM.

TABLA 5.9
Casos de prueba y valores esperados

Valores de entrada			Escenario	Resultados esperados
Sistema en línea	**Nombre de usuario**	***Password* de usuario**		
Si	dianatorres	diato12345	El computador tiene acceso al sistema. El sistema despliega cuadro de diálogo de acceso. El usuario digita nombre de usuario y password de usuario correctos.	El sistema habilita las opciones que el perfil de usuario puede utilizar. El sistema carga cuadro de diálogo principal de acceso. La variable de intentos de ingreso en 0. La BD cambia estado de usuario a "activo". El sistema bloquea segundo acceso para usuario.
No			El computador no tiene acceso al sistema.	El sistema despliega mensaje de error de conexión. La variable de intentos de ingreso en 0. Finaliza el proceso del caso de uso.
Si	dianatorres	diato1234	El computador tiene acceso al sistema. El sistema despliega cuadro de diálogo de acceso. El usuario digita nombre de usuario o password de usuario no válidos o incorrectos.	El sistema despliega mensaje de error. La variable de intentos de ingreso se incrementa en 1. *IF* variable de intentos = 3 *THEN* variable de intentos = 0 y fin del proceso *ELSE* el sistema solicita nuevamente nombre de usuario y password de usuario.

1.7 Validar el conjunto de casos de prueba

Algunas de las propuestas analizadas sólo validan la correctitud del recorrido de caminos, como *UML-based approach to system testing* y *Use case derived test cases*, pero no verifican la eficacia de los casos de prueba aplicados, por lo que el probador puede no haber seleccionado el criterio más adecuado u haya olvidado categorías de datos importantes. Además, la herramienta de soporte puede presentar fallos al momento de aplicar los casos de prueba, como sucedió con *Requirements by contracts allow automated system* cuando la herramienta quedó bloqueada en medio del proceso.

Otro concepto importante para validar los casos de prueba es determinar el momento de detención del proceso de la prueba, y para tomar esta decisión el probador debe conocer cuándo se ha alcanzado el grado de cobertura planificado, y si la especificación ha sido recorrida en su totalidad. Esta es una decisión difícil de planificar, debido a lo complicado que resulta conocer el número de casos de prueba para un determinado producto software, por lo que se recomienda tratarlo en un proyecto de trabajo futuro. La propuesta es utilizar un análisis previo, de parte de un probador experimentado, a las características que describen los casos de prueba, la cobertura planificada, los escenarios y los resultados esperados. La única propuesta que especifica cuándo parar el proceso de la pruebas es *Extended use case test design pattern*, que tiene en cuenta los valores esperados y alcanzados para los casos de prueba.

Para asegurar la calidad de la prueba y alcanzar los objetivos trazados, se verifican los resultados obtenidos en el proceso de aplicación de los casos de prueba. Esta verificación consiste en cuantificar si se hizo una adecuada cobertura de requisitos, si se obtuvo los resultados esperados y si se respetaron las reglas del negocio elicitadas. Si esa verificación es satisfactoria el conjunto de casos de prueba se considerará válido y de calidad; si no, se debe repetir el proceso de aplicación de cada uno de los pasos propuestos.

Dada la estructura incremental que tiene esta propuesta, se considera que el hecho de lograr cada uno de los pasos valida los casos de prueba, por lo que esta actividad consiste en chequear que se cumpla la cobertura, los valores de entrada, los escenarios y los valores esperados especificados en los casos de prueba. Es decir, al lograr avanzar de una actividad a la siguiente ya se está validando que la anterior se logró satisfactoriamente; por esto, el proceso mismo de lograr cada actividad es un indicador que valida los resultados de aplicación de los casos de prueba obtenidos al final del proceso de la prueba.

1.8 Generar el código de prueba

Esta actividad consiste en automatizar el proceso de diseño, selección y aplicación de los casos de prueba a un producto software. Aunque la propuesta *Requirements by contracts allow automated system* tiene una herramienta para hacerlo, presenta muchas dificultades de uso y aún no está terminada. Para realizar esta actividad es preciso que la especificación de requisitos se exprese formalmente, algo que se escapa a los objetivos de esta tesis y que se propone como trabajo futuro; por esto, la realización de esta actividad también se propone como investigación posterior.

REFERENCIAS

- Arango, I. F. & Zapata I. C. M. (2004). "Alineación entre metas organizacionales y elicitación de requisitos del software". *Dyna*, Vol. 71, No. 143, pp. 101-110.
- Arango, I. F. & Zapata J. C. M. (2006). "UN-Método para la elicitación de requisitos de software". Medellín: Universidad Nacional de Colombia.
- Binder, R. V. (2000). "Testing Object-Oriented Systems". USA: Addison-Wesley.
- Bohem, B. W. (1979). "Guidelines for verifying and validating software requirements and design specifications". *European conference on applied information technology of the international federation for information processing*. London, UK, pp. 711-719.
- Capuz, R. S., Gómez-Senent M. E., Torrealba L. A. & Ferrer G. P. (2000). "Cuadernos de ingeniería de proyectos III: dirección, gestión y organización de proyectos". España: Universidad Politécnica de Valencia.
- Davis, R. (2010). "What is a test scenario?" On line: http://www.robdavispe.com/free2/software-qa-testing-test-tester-2240.html, Agosto 2010.
- Dustin, E., Rashka J. & McDiarmid, D. (2002). "Quality Web Systems". USA: Addison-Wesley.
- Escalona, M. J. (2004). "Modelos y técnicas para la especificación y el análisis de la Navegación en Sistemas Software". *Ph.D. European Thesis*. Department of Computer Languaje and Systems, University of Seville. Seville, Spain.
- Ferriday, C. (2006). "Decision Table-Based Testing". *CS-339 Advanced Topics in Computer Science: Testing*. Department of Computer Science, Swansea University, West Glamorgan UK. December 1-3.
- Grassmann, W. K. & Tremblay J. P. (1998). "Matemática Discreta y Lógica". Madrid: Pearson Education.
- Jacobson, I. (1992). "Object Oriented Software Engineering: A Use Case Driven Approach". New York: Addison-Wesley Professional.
- Kaner, C. (2003). "An Introduction to Scenario Testing". *Florida Tech*, Florida, USA, June.
- Myers, G. J. (1979). "The Art of Software Testing". New York: John Wiley.
- Ostrand, T. J. & Balcer M. J. (1988). "The Category-Partition Method for specifying and generating functional tests". *Comunications of the ACM*, Vol. 31, No. 6, pp. 676-686.
- Riebisch, M., Philippow I. & Ilmenau M. G. (2003). "UML-Based Statistical Test Case Generation". *Lecture Notes In Computer Science*, Vol. 2591, pp. 394-411.
- Robinson, H. (2000). "Intelligent Test Automation". *Software Testing & Quality Engineering*, September-October, pp. 24-32.
- Saad, Y. (2003). "Iterative Methods for Sparse Linear Systems". USA: SIAM.
- Srivastava, V. K., Farr W. & Ellis W. (1997). "Experience with the Use of Standard IEEE 982.1 on Software Programming". *International Conference and Workshop*. Monterey, CA, USA, pp. 121-127.
- Vauthier, J-C (2007). "Decision tables: A testing technique using IBM Rational Functional Tester". Software Services, IBM, Software Group. Disponible en: http://www.ibm.com/developerworks/rational/library/jun06/vauthier/, Mayo 2010.

ANEXO I – Resumen de las técnicas de prueba funcionales

Técnica	Descripción
Acceptance testing	Prueba final basada en especificaciones de usuario o en el uso de los usuarios en un periodo definido de tiempo
Ad hoc testing	Similar a la prueba exploratoria, pero a menudo se espera que los probadores comprendan significativamente el software antes de probarlo
Alpha testing	Prueba del sistema cuando el desarrollo está por concluir; cambios menores de diseño se pueden introducir como resultado de estas pruebas. Por lo general lo hacen los usuarios finales u otros, pero no los desarrolladores o los probadores
Basis path testing	Pruebas de identificación basadas en el flujo y los caminos de un programa o sistema
Beta testing	Cuando el desarrollo y la prueba esencialmente se han terminado y se requiere encontrar los errores y fallas finales entes de la versión final del sistema. Típicamente las ejecutan usuarios u otros, pero no los desarrolladores o probadores
Black-box testing	Casos de prueba generados desde la funcionalidad del sistema
Bottom-up testing	Integración de módulos o programas de inicio desde lo fundamental
Boundary value testing	Casos de prueba generados desde los valores límite o clases de equivalencia
Branch coverage testing	Comprobar que cada rama tiene resultados verdaderos y falsos al menos una vez
Branch/condition coverage testing	Verificar que cada condición en una decisión tome todas las posibles salidas al menos una vez
Cause–effect graphing	Mapeado múltiple de entradas simultáneas que pueden afectar a otros, para determinar sus condiciones a probar
Comparison testing	Comparar las debilidades y fortalezas de software con los productos de la competencia
Compatibility testing	Probar lo bien que funciona el sistema en un entorno hardware/ software/sistema operativo/entorno de red particular
Condition coverage testing	Verificar que cada condición en una decisiones recorra todas las posibles salidas al menos una vez
CRUD testing	Construir una matriz CRUD -Create, Read, Update, Delete- para probar todos los objetos creados, leídos, actualizados y borrados
Database testing	Chequear la integridad de los valores de los campos
Decision tables	Tabla que muestra los criterios de decisión y sus acciones respectivas
Desk checking	Revisiones a la exactitud del código desarrollado
End-to-end testing	Es la visión "macro" de la prueba; implica probar el entorno completo del sistema en una situación que imita al mundo real, como la interacción con una base de datos, utilizar la red de comunicaciones, o la interacción con otros elementos de hardware, aplicaciones o sistemas, si procede
Equivalence partitioning	Cada condición de entrada se divide en dos o más grupos. Los casos de prueba se generan desde las clases validas e invalidas
Exception testing	Identificar los mensajes de error y los procesos de manipulación de excepción, y las condiciones que los provocan
Exploratory testing	A menudo se diseña una creativa e informal prueba de software que no se basa en planes de pruebas formales o casos de prueba; los probadores puede aprender del software a medida que lo prueban
Free-form testing	Ad hoc o intercambio de ideas mediante la intuición para definir casos de prueba
Gray-box testing	Una combinación de pruebas funcionales y estructurales para aprovechar las ventajas de ambas
Histograms	Una representación gráfica de los valores medidos organizados de acuerdo a la frecuencia de ocurrencia, utilizado para identificar los puntos calientes
Incremental integration testing	Pruebas continuas de una aplicación cuando se le agregan nuevas funcionalidades; requiere que los diversos aspectos de la funcionalidad de una aplicación se independicen para trabajarlos por separado antes que todas las partes del programa se completen, o que los conductores de pruebas los desarrollen según sea necesario; la ejecutan los desarrolladores o los probadores
Inspections	Revisión formal inter pares que utiliza listas de control, criterios de entrada, y criterios de salida
Integration testing	Pruebas de partes combinadas de una aplicación para determinar si juntos funcionan correctamente. Las "partes" pueden ser módulos de código, aplicaciones individuales, o aplicaciones cliente/servidor en una red. Este tipo de pruebas es especialmente relevante para sistemas cliente/servidor y distribuidos
JADs	Técnica que facilita a usuarios y desarrolladores disenar sistemas en sesiones conjuntas
Load testing	Probar una aplicación bajo cargas pesadas, como la prueba de un sitio Web en un rango de cargas para determinar en qué punto el sistema responde a tiempo, se degrada o falla
Mutation testing	Método para determinar si un conjunto de datos de prueba o de casos de prueba es útil, al introducir deliberadamente cambios en el código -"bugs"- y volverlo a probar con los datos y casos de la prueba original para determinar si se detectan los errores. Una implementación adecuada requiere grandes recursos de cómputo
Orthogonal array testing	Técnica matemática para determinar cuáles variaciones de los parámetros necesitan ser probadas

Pareto analysis	Analizar patrones de los defectos para identificar sus causas y fuentes
Performance testing	Término que a menudo se usa indistintamente con pruebas de estrés y de carga. Idealmente, las pruebas de rendimiento -y cualquier otro tipo de pruebas- se define en la documentación de requisitos, QA o los planes de pruebas
Positive and negative testing	Probar todos los valores de entrada positivos y negativos
Prior defect history testing	Crear casos de prueba o volverlos a ejecutar para todos los defectos encontrados en las pruebas previas del sistema
Prototyping	Enfoque general para recopilar datos de los usuarios para la construcción mediante la demostración de parte de una aplicación potencial
Random testing	Técnica que consiste en seleccionar aleatoriamente un conjunto específico de valores de entrada en el que un valor es tan probable como cualquier otro
Range testing	Para cada entrada, identificar el rango sobre el que el comportamiento del sistema debe ser el mismo
Recovery testing	Probar que tan bien se recupera un sistema de los accidentes, errores de hardware, u otros problemas catastróficos
Regression testing	Probar un sistema a la luz de los cambios realizados durante un desarrollo en espiral, una depuración, un mantenimiento, o el desarrollo de una nueva versión
Risk-based testing	Mide el grado de riesgo de negocio en un sistema para mejorar las pruebas
Run charts	Una representación gráfica de cómo una característica de calidad varía con el tiempo
Sandwich testing	Integrar módulos o programas desde el comienzo y el final simultáneamente
Sanity testing	Por lo general, es un esfuerzo de prueba inicial para determinar si una nueva versión del software funciona lo suficientemente bien como para aplicarle pruebas de mayor esfuerzo. Por ejemplo, si el nuevo software bloquea el sistema cada cinco minutos, y hace que funcione lentamente, o destruye bases de datos, el software puede no estar en las mejores condiciones como para justificar la realización de nuevas pruebas en su estado actual
Security testing	Probar que tan bien se protege el sistema contra accesos internos o externos no autorizados, los daño intencionales, etc., puede requerir técnicas de prueba sofisticadas
State transition testing	Técnica en la que primero se identifican los estados del sistema, a continuación se diseñan los casos de prueba para probar los factores que originan un cambio de estado
Statement coverage testing	Cada declaración en un programa se ejecuta al menos una vez
Statistical profile testing	Las técnicas estadísticas se utilizan para desarrollar un perfil de uso del sistema que ayuda a definir las rutas de transacción, las condiciones, las funciones y tablas de datos
Stress testing	Término usado a menudo como sinónimo pruebas de carga y de rendimiento. También se utiliza para describir tales pruebas como pruebas funcionales del sistema
Structured walkthroughs	Técnica para llevar a cabo una reunión en la que los participantes del proyecto examinar un producto del trabajo de los errores
Syntax testing	Técnica impulsada por datos para probar combinaciones de sintaxis de entrada
System testing	Pruebas tipo Caja Negra que se basa en la especificación de requisitos; cubre todas las partes combinadas de un sistema
Table testing	Pruebas de acceso, seguridad e integridad de los datos de entrada de la tabla
Test Case from Use Case	Los casos de prueba se diseñan a partir de los casos de uso y se valida la especificación de requisitos
Thread testing	Combinación de unidades individuales en hilos de funcionalidad que en conjunto realizan una función o conjunto de funciones
Top-down testing	Integración de módulos o programas a partir del comienzo
Unit testing	La escala más "micro" de la prueba; para probar determinadas funciones o módulos de código. Por lo general la realiza el desarrollador y no los probadores, ya que requiere un conocimiento detallado del diseño interno del programa y del código.
Usability testing	Pruebas de "facilidad de uso". Claramente, esto es subjetivo, y dependerá de la objetividad del usuario final o del cliente. Entrevistas con usuarios, encuestas, vídeos de las sesiones de usuario, y otras técnicas se pueden utilizar. Los desarrolladores y probadores normalmente no son adecuadas como probadores de usabilidad
User acceptance testing	Determinar si el software es satisfactorio para el usuario final o para el cliente
White-box testing	Los casos de prueba se definen examinado los caminos lógicos de un sistema

ANEXO II – Anomalías que detectan las técnicas para pruebas funcionales

Tipo	Técnica	Tipo de Anomalía	Detalle de la anomalía
Funcional	Boundary value testing Cause–effect graphing Comparison testing Compatibility testing Decision tables Exception testing Exploratory testing Free-form testing Histograms Incremental integration testing Mutation testing Orthogonal array testing Pareto analysis Performance testing Prototyping Random testing Range testing Run charts State transition testing Syntax testing Table testing Test Case from Use Case Usability testing User acceptance testing Domain testing Testing external interfaces States, state graphs, and transition testing Hardware simulator Careful integration of modules Brutal stress test Data-flow testing Transaction-flow testing	Requisitos Especificación Descripción Representación Secuencialidad Cambio de estado Formatos de salida Formatos de entrada Documentación Interfaces Tipo de variable Tipo de contenido Caminos perdidos	Es la descripción general de lo que hace el producto, y cómo debe comportarse –entradas /salidas–, y que puede ser incorrecta, ambigua, y/o incompleta: descripciones faltantes, incorrectas, incompletas o superfluas; cambios de estado en los datos almacenados en bases de datos; descripción de cómo la interfaz del software interactúa con el software externo, el hardware y los usuarios, tipos de parámetros incorrectos o inconsistentes, número u ordenación incorrecta de parámetros, diseño incorrecto, inexistente o poco claro de elementos, diseño incorrecto de los componentes de las interfaces, los sistemas externos, las bases de datos o los dispositivos de hardware, falta de mensajes apropiados, falta de mensajes de retroalimentación para el usuario.
Estructural	Basis path testing Bottom-up testing Branch coverage testing Branch/condition coverage Compatibility testing Condition coverage testing Database testing Desk checking Inspections Joint application designs Load testing Performance testing Recovery testing Sandwich testing Security testing Statement coverage testing Table testing Top-down testing Unit testing Inspections Automated data dictionaries Specification systems Sintax testing	Control y secuencia Lógicos Procesamiento Inicialización Flujo de datos y anomalías De codificación De datos De rutas lógicas Manejo de memoria Manejo del procesador Asignación de prioridades Ordenamiento de caminos en los buses del sistema De manejo del lenguaje De estructuras del lenguaje Caminos de código o lógica perdidos	Al describir estas anomalías se supone que el diseño detallado para los módulos de software, que se encuentra a nivel de pseudocódigo con los pasos de procesamiento, las estructuras de datos, los parámetros de las entradas/salidas y las principales estructuras de control, están definidos. El pseudocódigo contiene fórmulas mal especificadas, se producen divisiones por cero, ramificaciones tempranas, elementos de pseudocódigo inalcanzables, anidación impropia, procedimiento o llamadas a funciones inadecuadas, uso incorrecto de los operadores lógicos, un registro puede carecer de un campo, se le asigna un tipo incorrecto a una variable o a un campo de un registro, una matriz no puede tener el número adecuado de elementos asignados, o el espacio de almacenamiento se asigna de forma incorrecta
Revisión	Test debugging Test quality assurance Test execution automation Test design automation	De las pruebas	Los planes de prueba, casos de prueba, instrumento de prueba, y los procedimientos de prueba también puede contener errores: código del instrumento de prueba mal diseñado, implementado o probado.

ANEXO III – Taxonomía de las técnicas de prueba del software

Técnica	Clasificación					
	Manual	**Automática**	**Estática**	**Dinámica**	**Funcional**	**Estructural**
Acceptance testing	x	x		x	x	
Ad hoc testing	x				x	
Alpha testing	x			x	x	
Basis path testing		x		x		x
Beta testing	x			x	x	
Black-box testing		x		x	x	
Bottom-up testing		x		x		x
Boundary value testing		x		x	x	
Branch coverage testing		x		x		x
Branch/condition coverage		x		x		x
Cause–effect graphing		x		x	x	
Comparison testing	x	x		x	x	x
Compatibility testing	x	x				x
Condition coverage testing		x		x		x
Database testing		x		x		x
Decision tables		x		x	x	
Desk checking	x			x		x
End-to-end testing	x	x			x	
Equivalence partitioning		x		x		
Exception testing		x		x	x	
Exploratory testing	x			x	x	
Free-form testing		x		x	x	
Gray-box testing		x		x	x	x
Incremental integration testing	x	x		x	x	
Inspections	x		x		x	x
Integration testing	x	x		x	x	
JADs (joint application designs)	x				x	x
Load testing	x	x		x		x
Mutation testing	x	x		x	x	
Orthogonal array testing	x		x		x	
Pareto analysis	x				x	
Performance testing	x	x		x	x	x
Positive and negative testing		x		x	x	
Prior defect history testing	x		x		x	
Prototyping		x		x	x	
Random testing		x		x	x	
Range testing		x		x	x	
Recovery testing	x	x		x		x
Regression testing				x	x	
Risk-based testing	x		x		x	
Run charts	x		x		x	
Sandwich testing		x		x		x
Sanity testing	x	x		x	x	
Security testing	x	x				x
State transition testing		x		x	x	
Statement coverage testing		x		x		x
Statistical profile testing	x		x		x	
Stress testing	x	x		x		
Structured walkthroughs	x			x	x	x
Syntax testing		x	x	x	x	
System testing	x	x		x	x	
Table testing		x		x		x
Test Case from Use Case	x	x		x	x	
Top-down testing		x		x	x	x
Unit testing	x	x	x			x
Usability testing	x	x		x	x	
User acceptance testing	x	x		x	x	
White-box testing		x		x		x

ANEXO IV – APLICACIÓN DE LA PROPUESTA A UN ESTUDIO DE CASO

Actividad 1. Generar el modelo de comportamiento del sistema

Con base en la elicitación de requisitos especificados en casos de uso se genera el modelo ideal de comportamiento del sistema. **El primer paso** de esta actividad es diseñar *el diagrama de procesos* de la organización. Atendiendo la recomendación de Meyer (2000) no se detalla el diagrama de procesos con los sub-diagramas, ya que esto sería como implementar desde esta fase la especificación mediante diagramas de clase, algo para lo que no tiene información suficiente. La Fig. A.1 representa el diagrama de procesos de la organización. **El segundo paso** consiste en presentar el prototipo de un *cuadro de diálogo* que refleje la intención de la solución al problema, como el que representa la Fig. A.2. **El tercer paso** es especificar los requisitos elicitados en un *diagrama de casos de uso*, tal como aparece en la Fig. A.3.

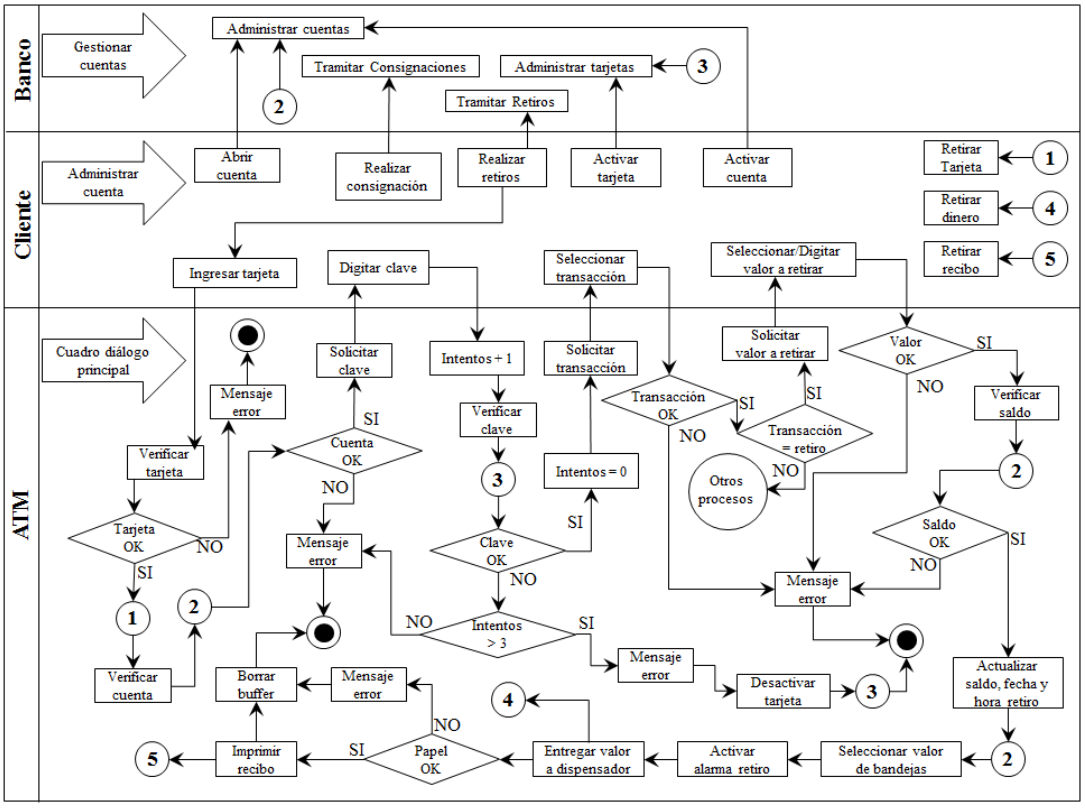

Fig. A.1 Diagrama de procesos de la organización

Fig. A.2 Cuadro de diálogo de la especificación

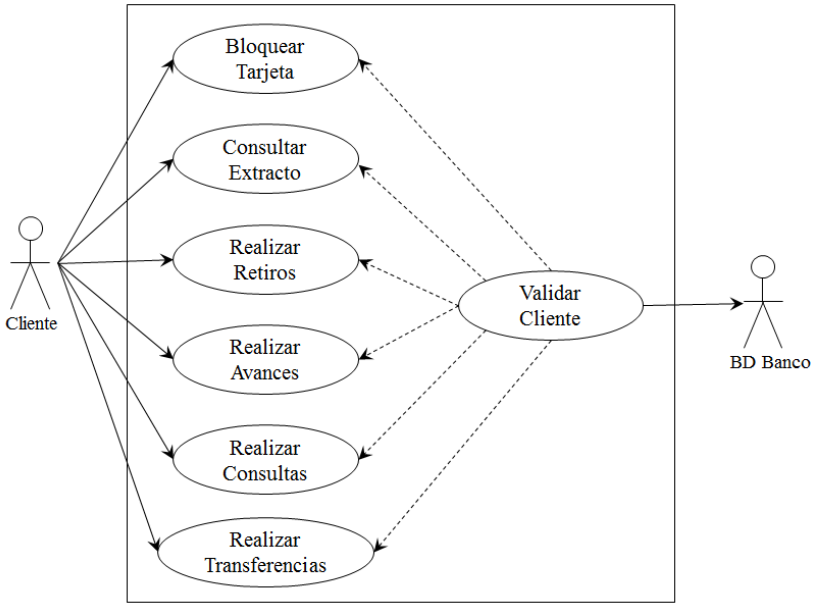

Fig. A.3 Diagrama de casos de uso del proceso

En el **cuarto paso** se representa la documentación que origina el proceso para diseñar los casos de prueba en una tabla. Esta tabla es un documento completo que constituye la base para realizar actividades como escenarios, valores de entrada, valores esperados, entre otras. Aunque la recomendación es que esta información debe presentarse lo más formaliza posible, en esta tesis no se tiene ese objetivo, por lo que en la Tabla A.1 se presenta lo más formalizada que se puede esperar de acuerdo con los alcances presentados para este trabajo.

Actividad 2. Diseñar el diagrama de caminos

El **primer paso** de esta actividad es extraer el listado de proposiciones desde el modelo de comportamiento y la tabla de documentación de los casos de uso. El concepto de proposición que se aplica es el de Grassmann & Tremblay (1998): "son afirmaciones que pueden ser verdaderas o falsas". El concepto de verdadero o falso representa el logro o no de las acciones que el actor realiza en el sistema en cada una de sus interacciones. La Tabla A.2 presenta las proposiciones de la especificación de la interacción *Realizar_Retiros*. Debido a que existen proposiciones que no se prueban debido a son acciones que llevan a cabo actores humanos, y a que otras están repetidas o se pueden fusionar en una sola, el **segundo paso** de esta actividad consiste en analizar las proposiciones mediante iteraciones hasta depurar el listado final. El **tercer paso** consiste en diseñar el grafo dirigido de caminos.

De la Tabla A.2 se eliminan las proposiciones 2, 4, 7, 11, 14, 21 y 25 del camino principal, por ser acciones humanas. El camino principal es el camino que toman los datos cuando la respuesta a las preguntas es afirmativa, y los alternos son los caminos que toman cuando la respuesta a esas preguntas es negativa; por esto se eliminan las proposiciones 1.1, 2.1, 3.1, 4.1, 5.1, 6.1, 7.1 y 8.1 de los caminos alternos, y se fusionan algunas para generar un sólo nodo, como lo propone el análisis de algoritmos (Skvarcius & Robinson, 1986) (Haggard *et al.*, 2006). Las proposiciones repetidas también se tienen en cuenta, y en las iteraciones se anulan dejando sólo la primera que documenta la tabla de proposiciones. En este caso se mantiene 1 y se anulan 27, 1.4, 2.4, 3.4, 4.5, 5.5, 6.5, 7.5 y 8.6; se fusiona 1.2 con 1.3 y se anulan las proposiciones que las repiten; se mantiene 5.4 y se anulan 6.4, 7.4 y 8.4; finalmente, se fusiona 6.2 con 6.3 y se anulan las repetidas.

TABLA A.1

Documentación de la especificación de la interacción *Realizar_Retiro*

Nombre Caso de Uso	Realizar transacciones en un ATM			
Código Caso de Uso	001			
Descripción Caso de Uso	El cliente utiliza un ATM para realizar operaciones bancarias			
Nombre Interacción	Realizar retiros			
Código Interacción	001-1			
Descripción Interacción	Un cliente retira dinero de su cuenta a través de un ATM			
Actor(es)	Banco, Cliente, ATM			
Pre-condiciones	ATM en buen estado, ATM en línea, ATM con dinero			
Dependencias anteriores	Abrir cuenta, cuenta y tarjeta activas, realizar consignación			
Camino Principal	**Acción Cliente**	**Regla del negocio**	**Acción/respuesta Sistema**	**Cambios de estado**
	1. Ingresar tarjeta	1. CALIDADTARJETA = SI	1. Verificar calidad tarjeta electrónicamente	
	2. Retirar tarjeta	2. Cuenta ∧ tarjeta activas	2. Verificar ESTADOCUENTA y TARJETA: BD	
			3. Cuadro de diálogo digitar CLAVE	
	3. Digitar CLAVE	3. CLAVE = 9999, INTENTOS < 3	4. Verificar CLAVE: BD	INTENTOS + = 1
			5. Cuadro de diálogo TRANSACCIÓN	INTENTOS = 0
	4. Seleccionar TRANSACCIÓN	4. TRANSACCIÓN selección opciones	6. Cuadro de diálogo VALORRETIRO	
	5. Seleccionar/digitar VALORRETIRO	5. VALORRETIRO (selección opciones ∨ múltiplo 10K)	7. Verificar VALORRETIRO	
		6. VALORRETIRO < SALDO	8. Verificar VALORRETIRO < SALDO: BD	
		6. SALDO = SALDO - VALORRETIRO	9. Actualizar SALDO: BD	SALDO = SALDO - VALORRETIRO
		7. FECHA = Date() ∧ TIME = Time()	10. Actualizar FECHA y HORA de retiro: BD	FECHA = Date(); TIME = Time()
			11. Seleccionar VALORRETIRO en billetes	Dinero disponible - VALORRETIRO
			12. Llevar VALORRETIRO a dispensador	
			13. Activar alarma dispensador de retiro	Alarma activada
	6. Retirar dinero		14. Verificar PAPEL	
			14. Imprimir recibo transacción	Alarma desactivada
	7. Retirar recibo		15. Borrar buffer	Valores a 0; Variables en blanco
Caminos Alternativos		1. CALIDADTARJETA = NO	1.1 Mensaje Error; FIN proceso	
		2. Cuenta ∨ Tarjeta desactivada	2.1 Mensaje error; FIN proceso	
		3. Clave incorrecta ∧ INTENTOS < 3	3.1 Mensaje error; FIN proceso	
		4. Clave incorrecta ∧ INTENTOS = 3	4.1 Mensaje error; FIN proceso	TARJETA = OFF
		5. TRANSACCIÓN incorrecta	5.1 Mensaje error; FIN proceso	INTENTOS = 0
		6. VALORRETIRO incorrecto	6.1 Mensaje error; FIN interacción Retirar Dinero	INTENTOS = 0
		7. VALORRETIRO > SALDO	7.1 Mensaje error; FIN interacción Retirar Dinero	INTENTOS = 0
		8. PAPEL = OFF	8.1 Mensaje error; FIN proceso	Valores a 0; Variables en blanco
Dependencias posteriores	Cambiar tarjeta; Reactivar cuenta; Reactivar tarjeta			
Post-condiciones	SALDO de cuenta actualizado; FECHA y HORA actualizadas; Valores a 0; Variables en blanco			
Autor	Edgar Serna M.			
Fecha modificación	Ninguna			
Observaciones	Ninguna			

TABLA A.2
Proposiciones de la interacción *Realizar_Retiros*

Camino	Proposiciones
Principal	1. El ATM despliega cuadro de diálogo principal
	2. El CLIENTE introduce la tarjeta en el ATM
	3. El ATM verifica calidad de la tarjeta
	4. El CLIENTE retira la tarjeta
	5. El ATM verifica estado de cuenta y tarjeta
	6. El ATM despliega cuadro de diálogo digitar clave
	7. El CLIENTE digita clave
	8. El ATM incrementa número de intentos en 1
	9. El ATM verifica clave
	10. El ATM despliega cuadro de diálogo seleccionar transacción
	11. El CLIENTE selecciona transacción
	12. El ATM verifica transacción
	13. EL ATM despliega cuadro de diálogo seleccionar/digitar valor retiro
	14. El CLIENTE selecciona/digita valor retiro
	15. El ATM verifica valor retiro
	16. El ATM verifica saldo de cuenta
	17. El ATM actualiza el nuevo saldo
	18. El ATM actualiza fecha y hora retiro
	19. El ATM selecciona de las bandejas los billetes de valor retiro
	20. El ATM activa la alarma retirar dinero
	21. El CLIENTE retira el dinero
	22. EL ATM desactiva la alarma retirar dinero
	23. El ATM verifica la existencia de papel
	24. El ATM imprime recibo transacción
	25. El CLIENTE retira recibo transacción
	26. El ATM iguala las variables al valor definido
	27. El ATM despliega cuadro de diálogo principal
Alternativo 1	1.1 La tarjeta está en mal estado
	1.2 El ATM despliega mensaje de error
	1.3 El ATM finaliza proceso
	1.4 El ATM despliega cuadro de diálogo principal
Alternativo 2	2.1 La cuenta o tarjeta está desactivada
	2.2 El ATM despliega mensaje de error
	2.3 El ATM finaliza proceso
	2.4 El ATM despliega cuadro de diálogo principal
Alternativo 3	3.1 La clave digitada es incorrecta y número de intentos es menor a 3
	3.2 El ATM despliega mensaje de error
	3.3 El ATM finaliza proceso
	3.4 El ATM despliega cuadro de diálogo principal
Alternativo 4	4.1 La clave digitada es incorrecta y el número de intentos es igual a 3
	4.2 El ATM despliega mensaje de error
	4.3 El ATM desactiva la tarjeta
	4.4 El ATM finaliza proceso
	4.5 El ATM despliega cuadro de diálogo principal
Alternativo 5	5.1 La transacción seleccionada es incorrecta
	5.2 El ATM despliega mensaje de error
	5.3 El ATM finaliza proceso
	5.4 El ATM iguala número de intentos a 0
	5.5 El ATM despliega cuadro de diálogo principal
Alternativo 6	6.1 El valor digitado para retirar es incorrecto
	6.2 El ATM despliega mensaje de error
	6.3 El ATM finaliza interacción Realizar Retiros
	6.4 El ATM iguala número de intentos a 0
	6.5 El ATM despliega cuadro de diálogo principal
Alternativo 7	7.1 El valor digitado para retirar es mayor al saldo de la cuenta
	7.2 El ATM despliega mensaje de error
	7.3 El ATM finaliza interacción Realizar Retiros
	7.4 El ATM iguala número de intentos a 0
	7.5 El ATM despliega cuadro de diálogo principal

Alternativo 8	8.1 El ATM no tiene papel
	8.2 El ATM despliega mensaje de error
	8.3 El ATM finaliza interacción Realizar Retiros
	8.4 El ATM iguala número de intentos a 0
	8.5 El ATM iguala las variables al valor definido
	8.6 El ATM despliega cuadro de diálogo bienvenida

Las iteraciones para eliminar proposiciones deben hacerse a través de un análisis con el cliente y el equipo de ingenieros de requisitos, ya que pueden existir proposiciones mal expresadas o que no reflejan las necesidades de los clientes, aunque para el equipo de pruebas reflejen algo que observa en la especificación. La Tabla A.3 presenta las proposiciones resultantes luego de las iteraciones, y la Fig. A.4 representa el grafo dirigido con las proposiciones de la Tabla A.2. Nótese que las aristas describen, entre otros, los requisitos no funcionales y la violación a las reglas del negocio, con que se representar una vista de la información de partida.

TABLA A.3
Proposiciones iteradas

Prioridad	Proposiciones
1	El ATM despliega cuadro de diálogo principal
2	El ATM verifica calidad de la tarjeta
3	El ATM verifica estado de cuenta y tarjeta
4	El ATM despliega cuadro de diálogo digitar clave
5	El ATM incrementa número de intentos en 1
6	El ATM verifica clave
7	El ATM despliega cuadro de diálogo seleccionar transacción
8	El ATM verifica transacción
9	EL ATM despliega cuadro de diálogo seleccionar/digitar valor retiro
10	El ATM verifica valor retiro
11	El ATM verifica saldo de cuenta
12	El ATM actualiza el nuevo saldo
13	El ATM actualiza fecha y hora retiro
14	El ATM selecciona de las bandejas los billetes de valor retiro
15	El ATM activa la alarma retirar dinero
16	El ATM desactiva la alarma retirar dinero
17	El ATM verifica la existencia de papel
18	El ATM imprime recibo transacción
19	El ATM iguala las variables al valor definido
20	El ATM despliega cuadro de diálogo principal
2.1	El ATM despliega mensaje de error y finaliza proceso
6.1	El ATM desactiva la tarjeta
8.1	El ATM iguala número de intentos a 0
10.1	El ATM finaliza interacción Realizar Retiros
17.1	El ATM despliega mensaje de error

Actividad 3. Determinar el criterio de cobertura y seleccionar caminos

El **primer paso** de esta actividad consiste en seleccionar el criterio de cobertura. Para esto se parte de la Fig. A.4 desde la que se calcula el conjunto de pares ordenados:

$$A = \left\{ \begin{array}{l} (I,1),\ (1,2),\ (2,2.1), (2.1,1), (2,3), (3,2.1), (3,4), (4,5), (5,6), (6,2.1), (6,6.1), (6.1, 2.1), (6,7), (7,8), (8,8.1), \\ (8.1,2.1), (8,9), (9,10), (10,10.1), (10.1,8.1), (10, 11), (11,10.1), (11,12), (12,13), (13,14), (14,15), (15,16), \\ (16,17), (17,17.1), (17,18), (18,19), (17.1,19), (19,1) \end{array} \right\}$$

Luego se halla la matriz binaria, como se representa en la Tabla A.4.

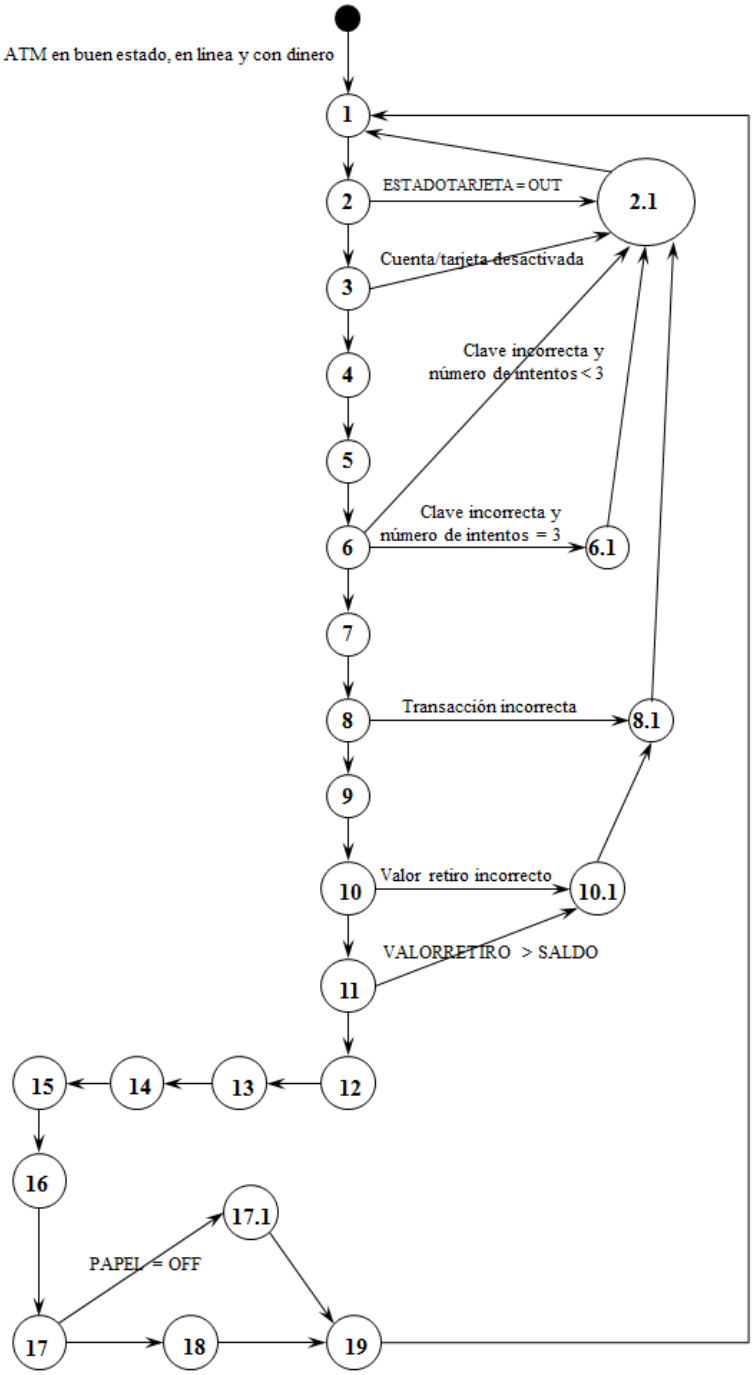

Fig. A.4 Grafo de caminos resultante

El **segundo paso** es seleccionar los caminos a recorrer de acuerdo con el grado de entrada y de salida de sus nodos, para el caso de estudio son:

1. (I,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,1)
2. (2,2.1,1)
3. (3,2.1,1)
4. (6,2.1,1)
5. (6,6.1,2.1,1)
6. (8,8.1,2.1,1)
7. (10,10.1,8.1,2.1,1)
8. (17,17.1,19)

TABLA A.4
Matriz binaria para el grado de entrada y salida de los nodos del grafo

	I	1	2	2.1	3	4	5	6	6.1	7	8	8.1	9	10	10.1	11	12	13	14	15	16	17	17.1	18	19
I		1																							
1			1																						
2				1	1																				
2.1		1																							
3				1		1																			
4							1																		
5								1																	
6				1					1	1															
6.1				1																					
7											1														
8												1	1												
8.1				1																					
9														1											
10															1	1									
10.1												1													
11															1		1								
12																		1							
13																			1						
14																				1					
15																					1				
16																						1			
17																							1	1	
17.1																									1
18																									1
19	1																								

Actividad 4. Seleccionar valores de entrada

Identificar las variables funcionales es el **primer paso** de esta actividad, el **segundo paso** es definir el dominio de esas variables y el **tercer paso** es identificar las relaciones operacionales. Para identificar las variables operacionales se consulta la Tabla A.1. Los datos fuente son fórmulas, cambios de estado, reglas de negocio y los que generan caminos alternativos. Para el estudio de caso se representa el primero y segundo pasos en la tabla A.5.

TABLA A.5
Variables operacionales y dominio

Variable	Domino	Tamaño
Calidad_Tarjeta	Booleano: Si/No	
Estado_Cuenta	Booleano: Si/No	
Estado_Tarjeta	Booleano: Si/No	
Clave	Numérico	4
Intentos	Numérico	1

Las variables Transacción, Valor_Retiro y Papel no se consideran operacionales dado que no son factores que varían entre las diferentes ejecuciones del caso de uso; a cambio son valores de una selección en un cuadro de diálogo que no influyen operacionalmente en el desarrollo de las interacciones, y sólo indican una posible ruta que debe tomar el programa ante un determinado valor seleccionado. La Tabla A.6 identifica las relaciones operacionales entre las variables de la Tabla A.5.

Actividad 5. Estructurar y refinar escenarios de prueba

Esta actividad se divide en dos pasos, en el **primer paso** se grafican los escenarios resultantes de acuerdo con la información suministrada en la Tabla A.1 de documentación de la especificación; y en el **segundo paso** se documentan en una tabla que representa al grafo del primer paso, las

variables operacionales y la valoración de las relaciones operacionales de la actividad anterior. El primer paso se detalla en la Fig. A.5 y el segundo en la Tabla A.7.

TABLA A.6
Relaciones operacionales resultantes

CalidadTarjeta	EstadoCuenta	EstadoTarjeta	Clave	Intentos
No	NA	NA	NA	NA
Si	No	No	NA	NA
Si	No	Si	NA	NA
Si	Si	No	NA	NA
Si	Si	Si	OK	NA
Si	Si	Si	NOT	< 3
Si	Si	Si	NOT	= 3

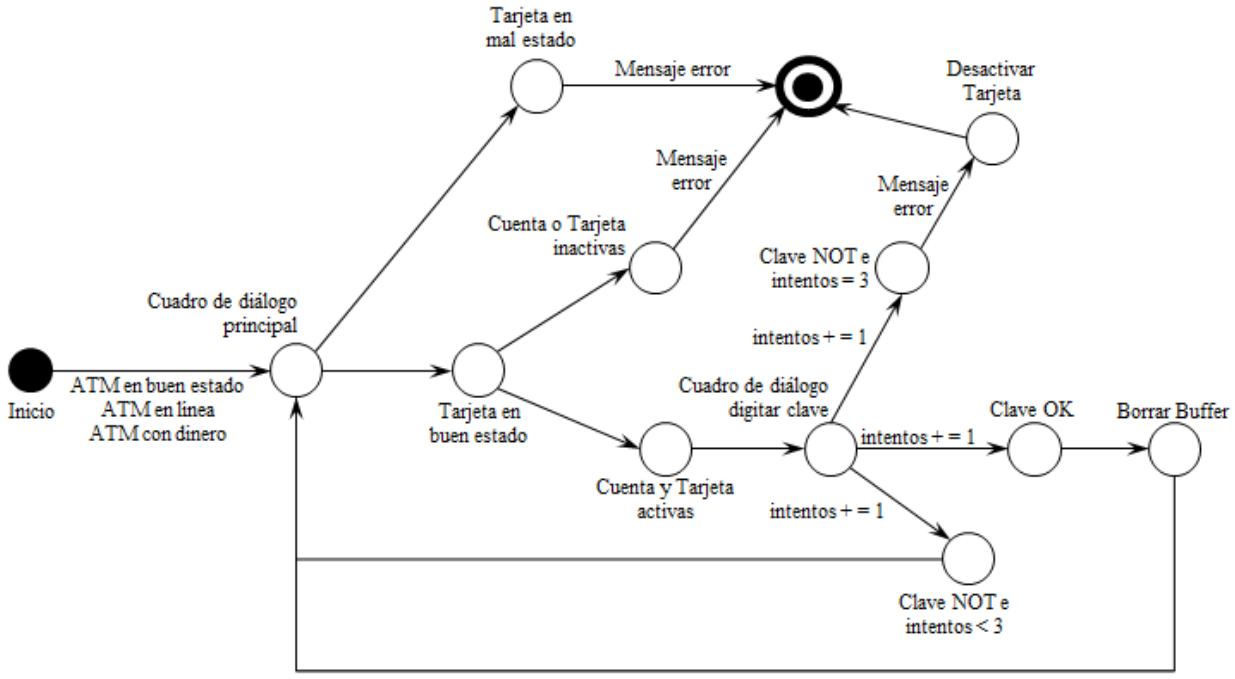

Fig. A.5 Grafo de escenarios del estudio de caso

Actividad 6. Calcular resultados esperados y diseñar casos de prueba

En la Tabla A.1 se elicitaron las reglas del negocio, las fórmulas y las excepciones que debe respetar el diseño de una posible solución al problema; esa información es la base para calcular los valores esperados al aplicar cada escenario, que se complementa para diseñar los casos de prueba. Por ejemplo, una regla de negocio básica en este estudio de caso es que el número de intentos para ingresar el *Password* siempre será menor a 3 antes que el sistema desactive la Tarjeta; o que la Cuenta y la Tarjeta deben estar activas para que el sistema permita seleccionar tipo de transacción. De acuerdo con esto, en la Tabla A.8 se presentan los casos de prueba estructurados desde los escenarios a los que se incorporan los valores calculados.

Actividad 7. Validar el conjunto de casos de prueba

Como se ha expresado antes, el hecho de ejecutar cada actividad mediante el proceso estructurado que aquí se propone es suficiente para esta propuesta para garantizar la validación del conjunto de casos de prueba, ya que partir de valores de entrada bien calculados y aplicar los escenarios de prueba es suficiente para lograr los resultados esperados. En el estudio de caso, cuando no se logra alguno de estos resultados es porque se ha fallado en alguno de los pasos de

las actividades anteriores, por lo que se recomienda revisar ese caso de prueba en particular y no todo el conjunto.

TABLA A.7
Escenarios documentados

Escenario	Calidad Tarjeta	Estado Cuenta	Estado Tarjeta	Clave	Intentos	Secuencia de acciones
Escenario del día feliz	Ok	Activa	Activa	Ok	≤ 3	Tarjeta en buen estado Cuenta y Tarjeta activas Clave correcta Sistema borra buffer Cuadro de diálogo principal
Tarjeta en mal estado	NOT	--	--	--	--	La Tarjeta está en mal estado. FIN
Cuenta o Tarjeta inactiva	Si	Inactiva	Activa	--	--	Tarjeta en buen estado Cuenta o Tarjeta inactiva FIN
Cuenta y Tarjeta acticvas	Si	Activa	Activa	NOT	< 3	Tarjeta en buen estado Cuenta y Tarjeta activas Clave incorrecta Intentos < 3 Cuadro de diálogo principal
Cuenta y Tarjeta activas	Si	Activa	Activa	NOT	= 3	Tarjeta en buen estado Cuenta y Tarjeta activas Clave incorrecta Intentos = 3 Sistema desactiva Tarjeta FIN

TABLA A.8
Casos de prueba estructurados

Escenario	Valores de entrada					Resultados esperados
	Calidad Tarjeta	Estado Cuenta	Estado Tarjeta	Clave	Intentos	
Tarjeta en buen estado Cuenta y Tarjeta activas	Ok	Activa	Activa	Ok	≤ 3	El Cliente Retira Dinero El Saldo está actualizado Fecha y Hora están actualizados El buffer está limpio ATM en cuadro de diálogo principal
La Tarjeta está en mal estado	NOT	--	--	--	--	ATM muestra mensaje de error ATM finaliza proceso ATM en cuadro de diálogo principal
Tarjeta en buen estado Cuenta o Tarjeta inactiva	Si	Inactiva	Activa	--	--	ATM muestra mensaje de error ATM finaliza proceso ATM en cuadro de diálogo principal
Tarjeta en buen estado Cuenta y Tarjeta activas Clave incorrecta	Si	Activa	Activa	NOT	< 3	ATM muestra mensaje de error ATM incrementa número de intentos ATM finaliza proceso ATM en cuadro de diálogo principal
Tarjeta en buen estado Cuenta y Tarjeta activas Clave incorrecta	Si	Activa	Activa	NOT	= 3	ATM muestra mensaje de error ATM desactiva tarjeta ATM finaliza proceso ATM borra buffer ATM en cuadro de diálogo principal

REFERENCIAS

- Meyer, B. (2000). "Object-Oriented Software Construction". New York: Prentice Hall.
- Grassmann, W. K. & Tremblay J. P. (1998). "Matemática Discreta y Lógica". Madrid: Pearson Education.
- Skvarcius, R. & Robinson W. B. (1986). "Discrete Mathematics with Computer Science Applications". California: Benjamin/Cummings.
- Haggard, G., Schlipf J. & Whitesides S. (2006). "Discrete Mathematics for Computer Science". USA: Thomson.