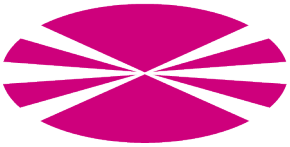
Aplicación de técnicas de pruebas automáticas basadas en propiedades a los diferentes niveles de prueba del software

---

Autor:
Miguel Ángel Francisco Fernández

Directora:
Laura M. Castro Souto

Tesis doctoral



UNIVERSIDADE DA CORUÑA
2015

Programa regulado por el RD 778/1998: Departamento de Computación

AGRADECIMIENTOS

No querría desaprovechar esta oportunidad sin agradecer a la persona que ha sido, sin duda, la más influyente en mi vida laboral, el que fue inicialmente el director de esta tesis y un gran amigo, Víctor M. Gulías, por ser una fuente de inspiración y motivación en todo momento para mí, por su ayuda, sus mensajes tranquilizantes y su apoyo, por su gran sabiduría y sus ideas brillantes, y, en especial, por lo mucho que he aprendido de él, tanto en lo académico, en lo profesional e incluso en lo personal. Gracias.

Me gustaría agradecer también a Laura M. Castro, directora de esta tesis, por toda la ayuda recibida durante todos estos años, comenzando con los primeros trabajos de investigación que hemos realizado juntos hasta la escritura de este trabajo. Sin ella, la realización de esta tesis no habría sido posible.

Gracias también a mis compañeros de trabajo de LambdaStream e Interoud, los cuales, a veces sin saberlo, me han ayudado mucho aportando información muy valiosa que ha podido ser usada en este trabajo acerca de la realización de las pruebas del software.

Y, por supuesto, muchas gracias a mis padres y a mi hermano, por su apoyo y ayuda constante a lo largo de estos años.

RESUMEN

Las pruebas son una de las actividades clave en el desarrollo de software, puesto que ayudan a detectar defectos que, de otro modo, pasarían desapercibidos hasta que el software sea desplegado. Sin embargo, al contrario que en otras etapas del ciclo de vida del software, como son el análisis, el diseño o la implementación, para las que existen metodologías y técnicas bien definidas y ampliamente aceptadas en la comunidad informática, junto con herramientas que permiten llevar a cabo dichas tareas, no hay una uniformidad sobre las metodologías, técnicas o herramientas a utilizar para llevar a cabo las pruebas del software de una manera eficiente y eficaz. Este hecho provoca que, muchas veces, éstas sean omitidas o no realizadas con todo el rigor necesario.

Esta tesis presenta una aproximación, basada en propiedades y puramente funcional, para la realización de las pruebas del software, que intenta paliar estos problemas. Para ello, se definen metodologías y técnicas de pruebas, integradas en el proceso de desarrollo de software, que pueden ser aplicadas a los diferentes niveles de pruebas del software. Así, pueden utilizarse para llevar a cabo pruebas unitarias y de componente, en las que se comprueba que cada componente individual se comporta de la manera esperada, pruebas de integración, que comprueban las interacciones de los componentes que forman parte de un sistema, y pruebas de sistema, que se encargan de comprobar diferentes aspectos del sistema como un todo. Además, se utiliza un lenguaje de especificación de pruebas común en todas las aproximaciones desarrolladas, el lenguaje de programación funcional Erlang, y las metodologías se definen de manera independiente a la estructura del software concreto a probar o el lenguaje de programación en el que éste esté implementado.

Por último, cabe destacar que el uso de estas metodologías y técnicas de pruebas se ilustra a través de un ejemplo industrial, en concreto, el sistema VoDKATV. Este sistema ofrece acceso a servicios multimedia (canales de televisión, videoclub, aplicaciones, juegos, entre otros) a través de diferentes tipos de dispositivos, como, por ejemplo, televisiones, ordenadores, tabletas o móviles. Con respecto a la arquitectura, el sistema VoDKATV está compuesto por múltiples componentes implementados con diferentes tecnologías (Java, Erlang, C, etc.) que se integran entre sí. La complejidad de este sistema permite ilustrar cada una de las metodologías y técnicas de pruebas desarrolladas con un ejemplo real.

SOBRE EL AUTOR

El contenido de esta tesis está ampliamente determinado por la actividad profesional de su autor durante la realización de la misma. Así, esta etapa ha estado marcada por el trabajo en la empresa LambdaStream, la cual, posteriormente, y después una serie de cambios, dio lugar a la empresa Interoud Innovation [34] en el año 2011. El propósito principal de la empresa Interoud Innovation, y anteriormente LambdaStream, es la construcción de sistemas interactivos para la televisión digital.

El principal producto de Interoud Innovation nace en el año 2006, dentro de la empresa LambdaStream, como resultado del proyecto fin de carrera del autor de esta tesis [185]. Este proyecto surge ante la necesidad de construir un sistema de televisión interactivo para hoteles. El sistema software resultante ha evolucionado a lo largo de estos años, adaptando su arquitectura y añadiendo nuevas funcionalidades a medida que eran requeridas por los clientes, además de poder ser utilizado en más tipos de entornos, no sólo hoteles, y desde más tipos de dispositivos, además de la televisión (ordenadores, teléfonos móviles o tabletas, entre otros). De la misma forma, el papel del autor de esta tesis con respecto a este sistema también ha evolucionado durante este tiempo, desde el perfil de programador en las etapas iniciales, hasta aumentar su responsabilidad en el ciclo de desarrollo, ejerciendo también tareas de arquitecto y líder de un equipo de trabajo formado por entre 3 y 5 personas.

Este sistema se conoce con el nombre de VoDKATV [35], y es, en la actualidad, un software complejo compuesto por múltiples componentes que se integran entre sí. Durante la realización de esta tesis, todas las metodologías y técnicas de pruebas desarrolladas se han utilizado para probar diferentes partes de este sistema. Es por ello que, en este trabajo, VoDKATV se usa como caso de estudio para ilustrar el funcionamiento de las metodologías y técnicas de pruebas desarrolladas.

Con respecto a la actividad investigadora del autor de esta tesis, cabe mencionar su vinculación al grupo de investigación MADS [42] desde el año 2008, cuyas actividades están relacionadas con los sistemas distribuidos y, en los últimos años, ha dedicado gran parte de los esfuerzos a la investigación en pruebas del software, lo cual se corresponde con la temática principal de esta tesis.

Además, es importante mencionar la participación en dos proyectos europeos relacionados con las pruebas del software, ambos del programa marco FP7. Por un lado, LambdaStream ha sido una entidad participante en el proyecto europeo ProTest [162] (2008-2011), en el que se han desarrollado técnicas y herramientas de pruebas basadas en propiedades. Por otro lado, con el rol de investigador principal dentro de la empresa Interoud Innovation, el autor de esta tesis ha participado en el proyecto europeo PROWESS [4] (2012-2015), cuyo principal objetivo es el desarrollo de técnicas y herramientas que faciliten las tareas de pruebas automáticas de servicios web, usando como base las aproximaciones basadas en propiedades desarrolladas durante el proyecto ProTest. Por esta razón, muchas de las metodologías y técnicas desarrolladas durante la realización de esta tesis se basan en actividades realizadas en estos proyectos, adaptando y especializando las aproximaciones de pruebas basadas en propiedades para ser usadas en diferentes niveles del software.

Finalmente, destacar que la realización de esta tesis ha dado lugar a los siguientes artículos y trabajos de investigación, los cuales han sido debidamente recopilados y organizados para producir el presente trabajo:

- **Applications integration: a testing experience** [137]. Laura M. Castro, Miguel A. Francisco, y Víctor. M. Gulías. *Proceedings of 6th Workshop on System Testing and Validation (STV'08)*, Madrid, España, 13 de Diciembre de 2008.
- **Testing Integration of Applications with QuickCheck** [139]. Laura M. Castro, Miguel A. Francisco, y Víctor M. Gulías. *Extended Abstracts of 12th International Conference on Computer Aided Systems Theory (EUROCAST'09)*, Las Palmas de Gran Canaria, España, 15-20 de Febrero de 2009.
- **A Practical Methodology for Integration Testing** [138]. Laura M. Castro, Miguel A. Francisco, y Víctor M. Gulías. *Computer Aided Systems Theory - EUROCAST'09. Lecture Notes in Computer Science*.
- **Property Driven Development in Erlang, by Example** [311]. Samuel Rivas, Miguel A. Francisco, y Víctor M. Gulías. *Proceedings of 5th Workshop on Automation of Software Test (AST'10)*, Cape Town, Sudáfrica, 1-8 de Mayo de 2010.
- **Uso de propiedades abstractas para especificación de pruebas funcionales de caja negra** [189]. Miguel A. Francisco, Laura M. Castro, y Víctor M. Gulías. *Actas de las XI Jornadas sobre Programación y Lenguajes (PROLE'11)*, A Coruña, España, 5-7 de Septiembre de 2011.
- **Automatic Generation of Test Models and Properties from UML Models with OCL Constraints** [187]. Miguel A. Francisco, y Laura M. Castro. *Proceedings of 12th Workshop on OCL and Textual Modelling (OCL'12, co-located with MODELS'12)*, Innsbruck, Austria, 1-5 de Octubre de 2012.

- **Automatización de Pruebas para Servicios Web: Generación de Propiedades y Modelos** [250]. Macías López, Henrique Ferreiro, Miguel A. Francisco, y Laura M. Castro. *Actas de las XIII Jornadas sobre Programación y Lenguajes (PROLE'13) y V Taller de Programación Funcional (TPF)*, Madrid, España, 17-20 de Septiembre de 2013.
- **Uso de propiedades y modelos para las pruebas de sistemas distribuidos basados en la integración de componentes heterogéneos** [186]. Miguel A. Francisco y Laura M. Castro. *Actas de las XIII Jornadas sobre Programación y Lenguajes (PROLE'13) y V Taller de Programación Funcional (TPF)*, Madrid, España, 17-20 de Septiembre de 2013.
- **Turning Web Services Descriptions into QuickCheck Models for Automatic Testing** [190]. Miguel A. Francisco, Macías López, Henrique Ferreiro, y Laura M. Castro. *Proceedings of 12th ACM SIGPLAN Workshop on Erlang (Erlang'13)*, Boston, Massachusetts, USA, 28 de Septiembre de 2013.
- **A Language-independent Approach to Black-box Testing Using Erlang As Test Specification Language** [136]. Laura M. Castro, y Miguel A. Francisco. *Journal of Systems and Software*, 2013.
- **Automatic Generation of Test Models for Web Services using WSDL and OCL** [253]. Macías López, Henrique Ferreiro, Miguel A. Francisco, y Laura M. Castro. *Proceedings of 11th International Conference on Service Oriented Computing (ICSOC'13)*, Berlín, Alemania, 2-5 de Diciembre de 2013. *Lecture Notes in Computer Science*.
- **Automating Property-based Testing of Evolving Web Services** [247]. Huiqing Li, Simon Thompson, Pablo Lamela Seijas, y Miguel A. Francisco. *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'14)*, San Diego, California, USA, 20-21 de Enero de 2014.
- **Model extraction and test generation from JUnit suites** [233]. Pablo Lamela Seijas, Simon Thompson, y Miguel A. Francisco. *University of Kent School of Computing Conference (UKSCC'15)*, Kent, Reino Unido, 15 de Junio de 2015.
- **Model extraction and test generation from JUnit suites** [234]. Pablo Lamela Seijas, Simon Thompson, y Miguel A. Francisco. Enviado a *7th International Symposium on Search-Based Software Engineering (SSBSE'15)*, Bérgamo, Italia, 5-7 de Septiembre de 2015.
- **A Property-based Load Testing Approach: a case study** [188]. Miguel A. Francisco, Laura M. Castro y Diana Corbacho. Enviado a *Journal of Systems and Software*, 2015.

Contenidos

FIGURAS

TABLAS

1

INTRODUCCIÓN

1.1. La importancia de las pruebas del software

Los sistemas software se han convertido en algo intrínseco y natural de la vida moderna. Las personas, cada vez, usan más sistemas software en el día a día, y en el mundo de la industria cada vez se dedica más dinero al desarrollo de software. El tamaño de los productos software ya no se mide en miles de líneas de código, sino en millones de líneas de código, y así, los sistemas software se han convertido en sistemas complejos que realizan todo tipo de tareas.

Por tanto, como los sistemas software son cada vez más importantes en cada una de las facetas de la sociedad, la demanda por la calidad de los mismos se incrementa. La calidad de los sistemas software depende en gran medida de las pruebas que hayan sido realizadas durante su desarrollo. Las pruebas son, por tanto, una de las actividades más importantes en el desarrollo de software. Sin ellas, el software contendría multitud de defectos que causarían fallos que únicamente podrían ser detectados cuando el producto se instale y se use en un entorno de producción real, provocando, por tanto, que el impacto de dichos fallos sea grave y el coste de solucionar los defectos que los causan sea grande [123].

Sin embargo, a pesar de la importancia de las pruebas, en muchas organizaciones no se les presta la suficiente atención o son, simplemente, omitidas en la planificación del desarrollo de productos software. Muchas veces, la presión por tener un producto terminado en una fecha establecida provoca que haya actividades, consideradas erróneamente como prescindibles, que son eliminadas o reducidas en la planificación. De esta forma, se persigue el objetivo de reducir costes y poder dis-

poner de un producto entregable a tiempo, aunque los inconvenientes suelen ser mucho mayores que las ventajas de entregar simplemente "algo". Así, eliminar las actividades de pruebas, en vez de acelerar el desarrollo de software, provoca que los productos software nunca puedan considerarse como realmente terminados, puesto que es posible que aparezcan continuamente defectos que deban ser resueltos una vez que el software haya sido desplegado. En este contexto, un defecto puede definirse como un comportamiento no esperado del sistema, el cual no se corresponde con los requisitos y especificaciones del mismo.

De esta forma, además de afectar directamente a la satisfacción de los clientes, el coste de solucionar estos defectos aumentará a medida que se avancen en las etapas del ciclo de vida del producto software. Esto se debe a que el defecto se ha ido propagando a lo largo de las diferentes etapas, y así, un defecto que podría ser corregido por un programador en un período corto de tiempo, puede requerir la interacción de más personas (programadores, probadores de software, el cliente, los usuarios, etc.) para poder informar, reproducir, solucionar y comprobar finalmente que el defecto ha sido corregido.

Según un estudio publicado por el NIST [343], el coste de solucionar un defecto aumenta hasta 30 veces cuando éste aparece después de desplegar el producto, frente a si se hace en las etapas iniciales del desarrollo (ver figura 1.1). Por otro lado, el *Systems Sciences Institute* de IBM calculó [215] que el coste de corregir un defecto encontrado en la etapa de mantenimiento es hasta cien veces mayor que el de eliminar un defecto encontrado en etapas iniciales (ver figura 1.2). Otros trabajos, como, por ejemplo, [124], también muestran resultados similares, en los que el coste de solucionar defectos aumenta conforme se avanza en cada una de las fases del ciclo de vida cuando se usa un modelo de desarrollo en cascada (ver figura 1.3).

Incluso, en muchas ocasiones, los defectos del software se acumulan hasta el punto que se deben identificar qué defectos deben ser arreglados inmediatamente y cuáles pueden esperar. Si esto perdura en el tiempo, es posible que el resultado final sea software frágil, delicado y poco mantenible, el cual puede ser visto por los programadores como algo difícil de modificar por temor a que, si se modifica, pueda haber otras partes del código fuente, para las que no exista ningún tipo de pruebas, que dejen de funcionar inesperadamente de la manera deseada. De hecho, la ausencia de un control de detección de errores que permita identificar, planificar y solucionar los defectos encontrados en el software, previniendo que éstos se multipliquen con el tiempo, es una de las causas habituales por las que los proyectos software fracasan [173, 221, 352].

Es por ello que las pruebas deben verse como una inversión. Así, los defectos ya corregidos no van a ser encontrados posteriormente y, por tanto, el tiempo empleado para corregirlos va a ser mucho menor que si éstos apareciesen más adelante. De esta forma, es posible concluir que, en el caso del desarrollo de software, *prevenir es mejor que curar*. Para ello, un conjunto de pruebas automáticas que permita

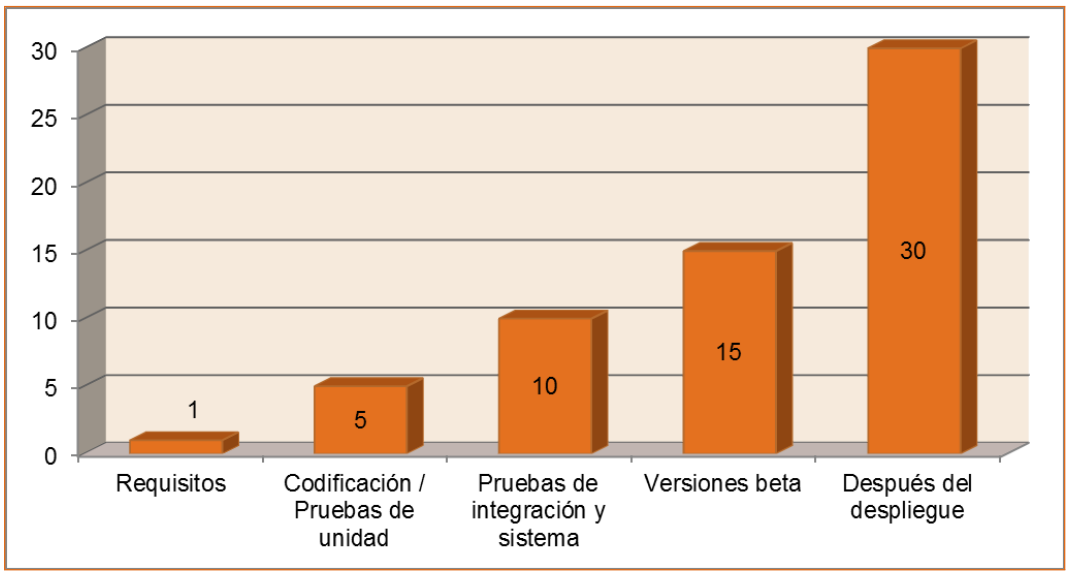


FIGURA 1.1: *Coste relativo de solucionar un defecto en función de la etapa de ciclo de vida en la que se encuentre [343]*

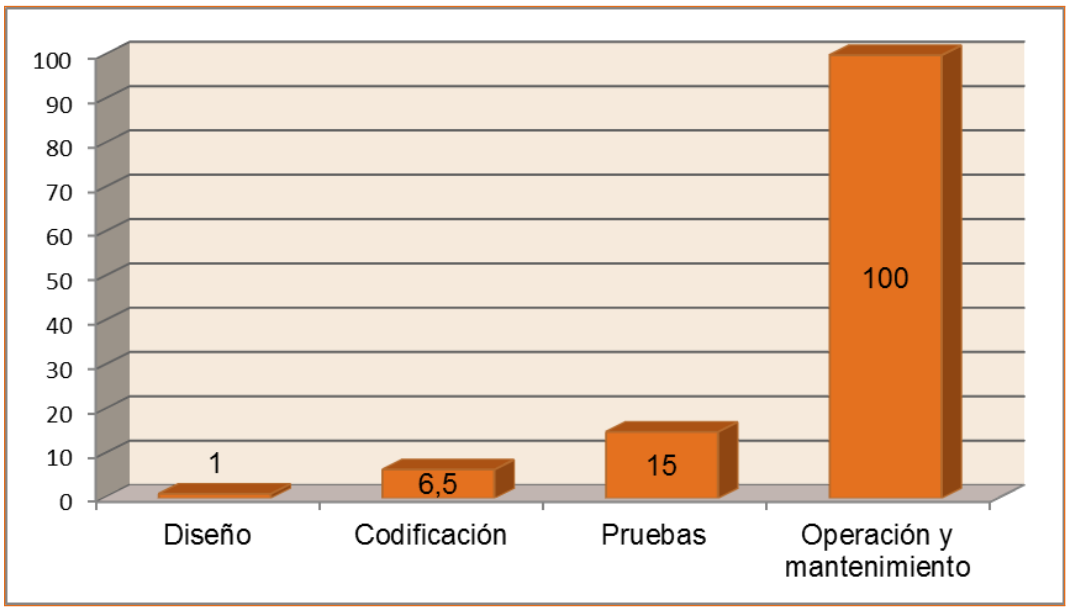


FIGURA 1.2: *Coste relativo de solucionar un defecto en función de la etapa de ciclo de vida en la que se encuentre [215]*

comprobar que el software cumple los requisitos establecidos permite llevar a cabo esta tarea.

Sin embargo, aunque está aceptado universalmente que las pruebas son muy im-

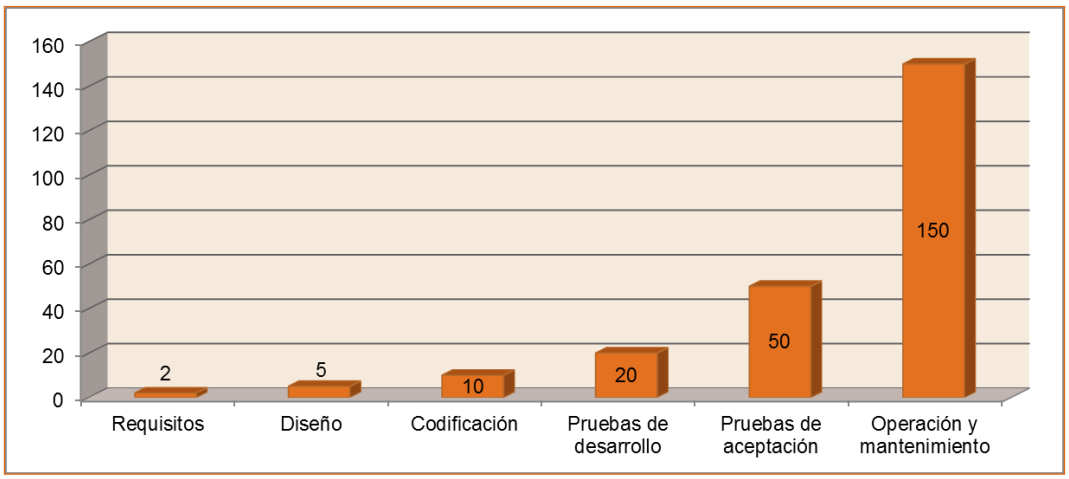


FIGURA 1.3: *Coste relativo de solucionar un defecto en función de la etapa de ciclo de vida en la que se encuentre [124]*

portantes y permiten aumentar la calidad del software, no es fácil encontrar una compañía en las que éstas se realicen con todo el rigor requerido. Muchas veces, las pruebas del software se pueden convertir en una tarea complicada [227, 347]. Por un lado, el diseño de algunos sistema software a probar no facilita la realización de las pruebas del mismo, en muchas ocasiones porque no se realizaron pruebas desde el comienzo de su implementación; y, otras veces, no se conocen las herramientas adecuadas para la realización de las pruebas, son difíciles y costosas de usar con casos reales, o no son lo suficientemente cómodas para el programador, que se ve obligado a escribir código de pruebas poco elegante y difícil de mantener [241, 304].

Asegurar que las pruebas se realicen, incluso teniendo en cuenta la dificultad que conlleva su implementación en algunos casos, es el gran reto de este trabajo. Para ello, se definirán metodologías y técnicas de pruebas que se integren dentro de un modelo de desarrollo que describirán, a través de casos de estudio que sirven como ejemplo de aplicación, cómo realizar las pruebas para que éstas sean eficientes en cuanto a la sencillez de su implementación y eficaces en cuanto a la detección de errores.

1.2. Motivación

Las pruebas del software son el método utilizado, tanto en el ámbito académico como en la industria, para verificar el comportamiento de los sistemas software. De esta forma, las tareas de pruebas juegan un papel muy importante en la implementación de cualquier aplicación. Además, el incremento de la complejidad de los sistemas software demanda que se usen herramientas que permitan probar el software de manera automática [322].

Así, una infraestructura de pruebas adecuada permitiría a los programadores encontrar defectos en el código antes y, por tanto, con menos esfuerzo y menor coste. Sin embargo, los métodos de pruebas empleados en la industria no son lo suficientemente avanzados como para que el software pueda desplegarse sin defectos o, con el menor número de defectos (a veces algunos conocidos) posible. Por ejemplo, existen marcos de desarrollo populares, como JUnit, que permiten automatizar las pruebas, pero al necesitar especificarse uno a uno los casos de prueba a usar, es necesario un esfuerzo bastante grande para probar de esta forma un sistema software complejo, además de que mantener dicho código de pruebas también se suele convertir en una tarea demasiado tediosa y costosa. En cualquier caso, a diferencia de otras etapas del ciclo de desarrollo software como el análisis, diseño o implementación, para las que existen una serie de herramientas y metodologías eficientes y eficaces, no existen estrategias y técnicas de pruebas que realmente causen un impacto importante en la industria.

Por otro lado, la presión por desplegar un producto software al mercado suele resultar en un deterioro de la calidad del software [93]. De hecho, se estima que los defectos que contiene el software por una infraestructura inadecuada de pruebas cuesta a la economía estadounidense un total de 59,5 miles de millones de dólares anualmente [343]. Además, se estima que se podrían ahorrar un total de 22,2 miles de millones de dólares anuales en Estados Unidos simplemente implementando una infraestructura de pruebas que permita identificar y solucionar defectos en el software [343].

Las actividades de pruebas son costosas y requieren tiempo para ser llevadas a cabo dentro del ciclo de desarrollo software [305]. De forma tradicional, se ha estimado que este tipo de actividades requiere entre un 30 % y 40 % del esfuerzo total de un proyecto típico de ingeniería del software, o incluso más en sistemas críticos [301]. De hecho, en algunas circunstancias, con los métodos de pruebas actuales, se ha observado que probar código puede requerir más tiempo y recursos que el propio desarrollo del software que está siendo probado [248]. Por su parte, el coste de los fallos del software en operación puede llegar a ser mucho mayor, e incluso éstos pueden llegar a ser catastróficos (se desintegró una sonda climática enviada a Marte por la NASA en el año 1999, se aplicaron dosis erróneas de radiación en el instituto nacional del cáncer de Panamá en el año 2000, el Airbus A380 se retrasó por problemas de incompatibilidad de software en el año 2006, se colapsó el aeropuerto de Los Ángeles en el año 2007, etc.). Es por ello que reducir el coste de las pruebas, manteniendo o incluso mejorando la efectividad de las mismas, es una de las líneas de investigación más relevantes tanto en la industria del software como en el ámbito académico.

Para asegurar la calidad del software, tanto los programadores como los probadores de software deben llevar a cabo una serie de objetivos, como son la localización de defectos de forma eficiente en etapas tempranas del desarrollo, ser capaz de so-

lucionar estos defectos antes de que el producto software sea desplegado y, por supuesto, mejorar las técnicas y herramientas de pruebas para que sea posible reducir el coste del desarrollo de software e incrementar la calidad del mismo [248]. De esta forma, la automatización de las pruebas, además de ahorrar tiempo dedicado a las mismas, permite mejorar la eficiencia en el proceso del desarrollo de software [350], acortando el tiempo de lanzamiento de nuevos productos al mercado, y con menos defectos que si fueran probados con pruebas manuales [210].

Todos estos datos han sido experimentados por el autor de este trabajo en su paso por la industria (tanto en la empresa LambdaStream como en Interoud Innovation). Durante esta etapa, la no realización de las pruebas o bien su realización de forma parcial ha ocasionado como resultado, en varias ocasiones, diversas entregas de productos software con numerosos defectos encontrados por primera vez cuando el software ha sido instalado en un entorno de producción. La falta de tiempo para realizar las pruebas siempre ha sido la mayor excusa para no realizarlas con toda la rigurosidad necesaria, incluso aunque, en muchos casos, su realización había sido contemplada en la planificación inicial del proyecto. Así, en muchas ocasiones, únicamente se realizaban pruebas manuales no muy exhaustivas o, en muchos otros casos, se implementaban manualmente *scripts* temporales que ejecutaban casos de prueba. En ambos casos, las pruebas realizadas no eran repetibles en futuras versiones del software y, por tanto, o bien se debía volver a invertir tiempo en preparar dichas pruebas, o bien funcionalidades ya implementadas no estarían probadas en nuevas versiones.

Es por ello que la motivación principal de este trabajo surge de esta situación, con el fin de desarrollar metodologías y técnicas de pruebas que realmente puedan ser llevadas a cabo por programadores y probadores para mejorar el proceso de desarrollo de software en la industria. Así, si es posible contar con las técnicas adecuadas para realizar las pruebas y con unas metodologías que describan cómo usar estas técnicas a lo largo del ciclo de desarrollo de un producto software, junto con las herramientas adecuadas que permitan llevarlas a cabo, también puede ser posible que las pruebas sean realmente realizadas de forma eficiente y eficaz. De esta forma, además de evitar los problemas descritos anteriormente cuando no se realizan las pruebas del software de la manera adecuada, es posible beneficiarse de las ventajas que suponen la realización de las pruebas sobre el desarrollo de software.

Por esta razón, a lo largo de este trabajo se desarrollarán metodologías y técnicas de pruebas con este fin. Además, como se podrá observar, todas las metodologías y técnicas de pruebas descritas en los diferentes capítulos han sido puestas en práctica por el autor de este trabajo en su entorno profesional en la industria, con la principal motivación de intentar mejorar las carencias existentes comentadas anteriormente en relación con las tareas de pruebas.

1.3. Objetivos

El principal objetivo de este trabajo es la definición de un conjunto de metodologías y técnicas de pruebas que puedan ser usadas a lo largo del ciclo de vida de un producto software. Estas metodologías y técnicas de pruebas estarán adaptadas según la etapa del ciclo de vida en la que se encuentra el producto software (implementación de componentes unitarios, integración de varios componentes, etc.), pero siempre tendrán como objetivo principal conseguir que las pruebas sean más eficientes, es decir, invertir menos tiempo en la realización de las mismas, y más efectivas, esto es, encontrar más defectos lo más pronto posible en el desarrollo. De esta forma, la idea principal es que, usando este tipo de técnicas, se aumente la calidad de los productos software.

Todas las técnicas de pruebas descritas en este trabajo están basadas en una aproximación funcional basada en propiedades. Este tipo de aproximaciones, conocidas como *pruebas basadas en propiedades* (PBT) [162], reemplazan el uso de casos de prueba individuales, escritos manualmente, por propiedades y modelos de alto nivel a partir de los cuales dichos casos de prueba son generados automáticamente. De esta forma, este tipo de técnicas conllevan una reducción importante de líneas de código con respecto a los métodos tradicionales en los que se especifican cada uno de los casos de prueba manualmente. Además, a parte de ayudar a descubrir más defectos en el software, incrementando la confianza en el funcionamiento deseado del sistema, las aproximaciones basadas en propiedades reducen el tiempo dedicado a las pruebas, reduciendo, de esta forma, el coste de los productos software y el tiempo necesario para que puedan estar disponibles en el mercado.

Por otro lado, este trabajo busca que las técnicas de pruebas descritas puedan ser usadas para probar diferentes tipos de sistemas, no únicamente sistemas implementados con un lenguaje de programación concreto. De esta forma, se evita que sea necesario utilizar aproximaciones diferentes en función del lenguaje de programación usado, y se consigue que las personas que usen estas técnicas puedan especializarse en el uso de las mismas, puesto que no están ligadas a probar sistemas escritos en un lenguaje de programación específico.

De la misma forma, para conseguir que las técnicas de pruebas descritas sean uniformes, cada una de las aproximaciones descritas para probar los distintos aspectos de un sistema software (componentes individuales, APIs de integración, integraciones entre componentes, etc.) usarán un mismo lenguaje de especificación de pruebas, evitando el uso de diferentes lenguajes para especificar el comportamiento del *sistema a probar* (SUT). El lenguaje elegido para este propósito ha sido el lenguaje de programación Erlang [84, 85] , el cual es un lenguaje de programación funcional, desarrollado por la compañía sueca Ericsson, con soporte específico para la implementación de sistemas distribuidos, tolerantes a fallos y con requisitos de tiempo real. Además, Erlang es una plataforma portable que se ejecuta en una máquina virtual y, por tanto, puede ser usado en diferentes sistemas operativos.

El uso del Erlang como lenguaje de especificación de pruebas está motivado por su naturaleza funcional y su sintaxis declarativa, lo cual lo convierte en un buen lenguaje de especificación para escribir propiedades y modelos legibles y concisos. Además de ser un lenguaje potente, expresivo y de alto nivel, Erlang proporciona interfaces con otros lenguajes de programación, como C o Java, por lo que permite su integración de forma sencilla con implementaciones escritas en otros lenguajes de programación. Así, esta integración permite usar Erlang como lenguaje de especificación de pruebas incluso si el sistema a probar está implementado en otro lenguaje de programación, puesto que es posible ejecutar casos de prueba escritos o generados desde Erlang contra implementaciones de sistemas escritas en lenguajes diferentes al propio Erlang.

Por último, debido a que las técnicas de pruebas estarán basadas en el uso de propiedades para generar casos de pruebas, el uso de Erlang tiene como ventaja que es posible usar librerías y herramientas ya existentes creadas específicamente para realizar pruebas basadas en propiedades. En particular, en este trabajo se usará la versión Erlang de QuickCheck [144] implementada por Quviq [58], una herramienta de pruebas automática basada en propiedades, tanto para definir especificaciones de pruebas como para generar y ejecutar casos de prueba.

En resumen, definir una serie de metodologías y técnicas de pruebas, basadas en propiedades, que puedan ser usadas en los diferentes niveles de pruebas del software a lo largo de las fases del ciclo de vida de un producto software, utilizando siempre un lenguaje común de especificación de pruebas, independiente del lenguaje de programación en el que esté implementado el sistema a probar, es el principal objetivo de este trabajo.

1.4. Estructura y contenidos

Este documento está dividido en capítulos. Así, después de este capítulo de introducción, el capítulo 2 explica los conceptos básicos que se tomarán como base a lo largo de este trabajo, definiendo lo que es el proceso de desarrollo de software y los diferentes ciclos de vida que pueden usarse en dicho proceso. También se explica el papel de las pruebas del software en los ciclos de vida del software, y se describen métodos, técnicas y herramientas de pruebas ya existentes, centrándose en las pruebas basadas en propiedades, puesto que representan las bases de las técnicas explicadas a lo largo de este trabajo.

A continuación, en el capítulo 3, se describe el caso de estudio usado para ilustrar con ejemplos las metodologías y técnicas de pruebas explicadas a lo largo de este trabajo. Dicho caso de estudio es un sistema real desarrollado en la industria, cuyo nombre es VoDKATV. Este sistema integra diferentes componentes para poder ofrecer a los usuarios finales una experiencia multimedia en múltiples dispositivos, como, por ejemplo, televisiones, ordenadores, tabletas o móviles, proporcionando

acceso a canales de televisión, películas, aplicaciones o juegos, entre otros. Gracias al uso de este sistema, además de ilustrar cómo se usan las metodologías y técnicas descritas, ha sido posible evaluar dichas propuestas con un sistema real.

El sistema VoDKATV se tomará, por tanto, como ejemplo de sistema software complejo, compuesto por diferentes componentes, implementados con diferentes lenguajes de programación, y que se integran entre sí para ofrecer una serie de funcionalidades. Este sistema se usará como ejemplo en los siguientes capítulos, los cuales describen el papel de las pruebas en las diferentes fases del desarrollo de un producto software:

- El capítulo 4 realiza una breve introducción a las fases de análisis y diseño de sistemas software, explicando los lenguajes de modelado usados para realizar este tipo de tareas, así como la relación entre las tareas de análisis y diseño con la posterior realización de las pruebas del software.
- Posteriormente, en el capítulo 5 se describen técnicas de pruebas a utilizar en la implementación de componentes individuales, los cuales surgen del análisis y diseño previo.
- Puesto que algunos componentes software se integrarán con otros componentes a través de APIs de integración, el siguiente capítulo, el capítulo 6, explica cómo realizar las pruebas de dichas APIs de integración.
- Como un caso particular y muy común de API de integración es el uso de un servicio web, se dedicará el capítulo 7 a explicar técnicas específicas para este caso.
- A continuación, el capítulo 8 describirá técnicas a usar para probar la integración propiamente dicha de dos componentes previamente probados de forma independiente.
- Por último, el capítulo 9 se centra en probar un sistema completamente integrado, realizando una introducción sobre cómo pueden ser aplicadas las pruebas basadas en propiedades para realizar pruebas no funcionales, en concreto, pruebas de rendimiento.

Finalmente, el capítulo 10 expone las conclusiones finales de este trabajo, junto con las lecciones aprendidas y las líneas de investigación abiertas. El contenido de este capítulo está basado en los resultados obtenidos de aplicar las diferentes metodologías y técnicas de pruebas descritas a lo largo de los capítulos anteriores al sistema real VoDKATV usado como caso de estudio.

2

CONTEXTUALIZACIÓN

2.1. Introducción

El término *software* puede definirse como el conjunto de programas, procedimientos, documentación asociada y datos relacionados con el funcionamiento de un sistema informático; en contraposición con el *hardware*, que es el equipamiento físico usado para procesar, almacenar y transmitir programas informáticos o datos [1]. Por otro lado, un *sistema software* es una colección de componentes organizados para llevar a cabo una función específica o conjunto de funciones. Cabe destacar que, en este trabajo, se hará referencia a este concepto usando simplemente el término *sistema*, utilizando explícitamente el término *sistema hardware* cuando es necesario referirse al *hardware* físico.

El mundo del software ha estado en continua evolución desde sus comienzos en los años 40, cuando el software comenzó siendo considerado como un complemento para llevar a cabo algunas tareas específicas, hasta la actualidad, en la que el software es una parte intrínseca de la sociedad. Durante este tiempo, ha pasado por diferentes etapas, entre las que destaca la vivida a mediados de los años 60, denominada comúnmente como *crisis del software* [167], término que se usaba para describir los frecuentes problemas que aparecían durante el proceso de desarrollo de nuevo software. El origen de esta etapa se debió a que la programación se consideraba un "arte", para la que no existían metodologías, ni se realizaba ningún tipo de planificación. Esto conllevaba retrasos importantes en las entregas, poca productividad, elevadas cargas de mantenimiento, baja calidad y fiabilidad del software, así como dependencia de los programadores, entre otros aspectos.

Este hecho dio popularidad al uso del término *ingeniería del software* [328], el cual fue introducido por Fritz Bauer en la primera conferencia sobre desarrollo de software patrocinada por el Comité de Ciencia de la OTAN celebrada en Garmisch (Alemania) en octubre de 1968. Como crear software se trataba como un proceso creativo, se buscaba sistematizar este proceso con el fin de acotar el riesgo del fracaso en la consecución del objetivo, por medio de diversas técnicas que demostraron ser adecuadas en base a la experiencia.

Actualmente, la ingeniería del software se define como la aplicación de un enfoque disciplinado, sistemático y cuantificable al desarrollo, operación y mantenimiento del software, es decir, la aplicación de la ingeniería al software; así como el estudio de dichos enfoques [1]. La ingeniería del software es una nueva rama de la ingeniería que crea y mantiene las aplicaciones de software usando tecnologías y prácticas de las ciencias de la computación, manejo de proyectos, el ámbito de la aplicación, y otros campos.

Así, como toda ingeniería, la ingeniería del software aplica *procesos*, *métodos* sistematizados y *herramientas* preestablecidas para resolver problemas, teniendo en cuenta las diferentes soluciones para elegir la más apropiada, siempre apoyándose en un compromiso con la calidad [301]. Mientras los procesos definen un marco de trabajo base para el control de la gestión de proyectos de software y establecen el contexto en el que los métodos pueden ser aplicados; los métodos o técnicas describen procedimientos formales para obtener ciertos resultados, que normalmente se llevan a cabo con la ayuda de herramientas específicas.

Existen diferentes entidades (universidades, organismos de normalización o investigación, empresas, etc.) que han abordado el análisis de problemas en el desarrollo de software, publicando textos didácticos, normativos o estándares para abordar el desarrollo, mantenimiento y operación con las mayores garantías de éxito. Entre estas entidades destacan por su mayor reconocimiento internacional ISO [37], IEEE Computer Society [33] y SEI [61].

La ingeniería del software abarca las diferentes actividades del desarrollo de software, desde la planificación, especificación de requisitos, el modelado del software (análisis y diseño), la codificación y pruebas del mismo, así como el mantenimiento y las optimizaciones que se deban realizar en el software a lo largo de su vida útil. En este capítulo se presentan algunos de estos conceptos, específicamente, aquellos que son necesarios para entender los objetivos y relevancia de este trabajo, centrándose en las actividades de pruebas. Así, la sección 2.2 describe en qué consiste el proceso de desarrollo de software y, posteriormente, en la sección 2.3, se ofrece una visión general de los ciclos de vida del software existentes. A continuación, en la sección 2.4, se realiza una introducción a las pruebas del software, explicando conceptos generales como la diferencia entre la verificación y la validación, los niveles de pruebas y diferentes tipos de técnicas de pruebas categorizadas según diferentes criterios. Finalmente, la sección 2.5 se centra en introducir las pruebas basadas en

propiedades, puesto que es la técnica de pruebas más relevante en la que se basa este trabajo.

2.2. El proceso de desarrollo de software

El *proceso de ingeniería del software* se define como un conjunto de actividades estructuradas que se deben llevar a cabo con la intención de lograr un objetivo, en este caso, la obtención, dentro de unos límites temporales, de un producto software con la calidad suficiente para satisfacer tanto a los clientes que han solicitado su creación, como a los usuarios que van a utilizarlo [301].

Cabe destacar que el proceso de ingeniería del software no es algo rígido, sino que es una aproximación que puede ser adaptada para que se elijan las actividades más apropiadas para construir, en cada caso, el software concreto. Sin embargo, hay cuatro actividades fundamentales que son comunes a todo proceso de desarrollo de software para la creación de un nuevo producto software [328]:

- **Especificación**: en las actividades de especificación y conceptualización (siempre presentes en el inicio de cualquier sistema software) se establecen los objetivos globales que debe cumplir el software, así como sus propiedades, características, restricciones (como, por ejemplo, rendimiento o usabilidad) y funcionalidades que deben estar presentes. Para ello, los usuarios y los ingenieros de software definen el software a producir, a través de un proceso de análisis y definición de requisitos [329, 330].
- **Desarrollo**: esta es la actividad principal en el desarrollo de un proyecto software [118, 204], y tiene como objetivo convertir una especificación en un sistema ejecutable. Además, puede requerir que la especificación de requisitos sea refinada si se usa un enfoque incremental. Estas actividades se dividen en dos etapas generales: el diseño del sistema propuesto (incluyendo su arquitectura, entorno o estructura), y su implementación en una serie de plataformas concretas y usando lenguajes de programación concretos.
- **Validación**: estas actividades evalúan si el software es adecuado, es decir, si cumple su especificación así como las expectativas del cliente. Las actividades de validación pueden ser llevadas a cabo desde diferentes niveles de pruebas [117, 152]. Así, es posible aplicarlas a componentes individuales (pruebas de unidad y pruebas de componente), a las interacciones entre diferentes componentes (pruebas de integración), al sistema completo (pruebas de sistema), y al cumplimiento de las expectativas por parte de los usuarios (pruebas de aceptación). Además, las inspecciones y revisiones también forman parte de este tipo de actividades.
- **Evolución**: la vida de un producto software no termina cuando está preparado para funcionar en un entorno de producción y se entrega al cliente [300].

Después de su despliegue, es muy probable que se requieran actividades de mantenimiento y soporte, incluyendo no sólo resolver problemas menores que podrían aparecer, sino también la mejora y evolución del sistema en caso de que surjan nuevas necesidades. Por tanto, el software debe poder ser modificado para adaptarse a cambios en el mercado y a las necesidades de los usuarios.

Aunque los procesos software suelen incluir otras actividades, como pueden ser la gestión de riesgos, actividades de seguimiento de proyectos, o actividades de gestión de configuración, entre otras, las cuatro actividades fundamentales descritas anteriormente siempre forman parte de un proceso de desarrollo de software de alguna forma u otra. Por su parte, estas actividades adicionales suelen formar parte de las cuatro actividades principales descritas anteriormente.

En resumen, el proceso de desarrollo de software involucra todas las actividades por las que las necesidades del usuario se transforman en un producto software [1]. Para ello, las necesidades del usuario son traducidas en requisitos del software, los cuales son transformados en un diseño y, el diseño, a su vez, es implementado en código. Este código es probado, documentado y, a veces, se prepara para su uso operacional y se instala en el entorno del cliente. Estas actividades se pueden superponer o incluso ser llevadas a cabo de manera iterativa hasta conseguir el producto software objetivo.

2.3. Ciclos de vida del software

El *ciclo de vida del software* se refiere al período de tiempo que transcurre desde que un producto software nace, hasta que dicho software se retira o reemplaza, dejando de estar disponible para su uso [1]. Las funciones principales de un ciclo de vida son las de organizar y estructurar un conjunto de actividades y procesos que permiten dirigir el desarrollo de un producto software. Para ello, un ciclo de vida define una serie de fases, para las cuales establece su orden, los criterios para pasar de una fase a la siguiente, así como las entradas y salidas de cada fase. Un ciclo de vida también debe describir los estados por los que pasa un producto, así como las actividades a realizar en cada uno de ellos sobre dicho producto.

Esta definición contrasta con la de *ciclo de desarrollo del software*, que se refiere al período de tiempo que comienza con la decisión de desarrollar un producto software y finaliza cuando el software es entregado [1]. Así, mientras que el ciclo de vida del software normalmente incluye las fases de conceptualización, requisitos, diseño, implementación, pruebas, instalación y puesta en marcha, operación y mantenimiento y, a veces, la retirada del producto; el ciclo de desarrollo del software únicamente suele incluir las fases de requisitos, diseño, implementación, pruebas y, a veces, instalación y puesta en marcha. Todas estas fases podrían superponerse en el tiempo o repetirse iterativamente.

Existen una serie de *modelos de ciclo de vida del software*, los cuales definen marcos de trabajo que contienen las actividades involucradas en el desarrollo, explotación y mantenimiento de un producto software, abarcando toda la vida del mismo, desde la definición hasta la finalización de su uso. La elección de un modelo de ciclo de vida u otro depende del tipo de proyecto y de la metodología elegida para la realización del mismo.

El primer modelo de ciclo de vida del software fue formalizado por Winston W. Royce en el año 1970 [313]. Este modelo, conocido con el nombre de *modelo en cascada*, consiste en una serie de fases totalmente secuenciales que se deben llevar a cabo de forma lineal, en concreto: extracción de requisitos del sistema, extracción de requisitos del software, análisis, diseño, codificación, pruebas, operación y mantenimiento (ver figura 2.1).

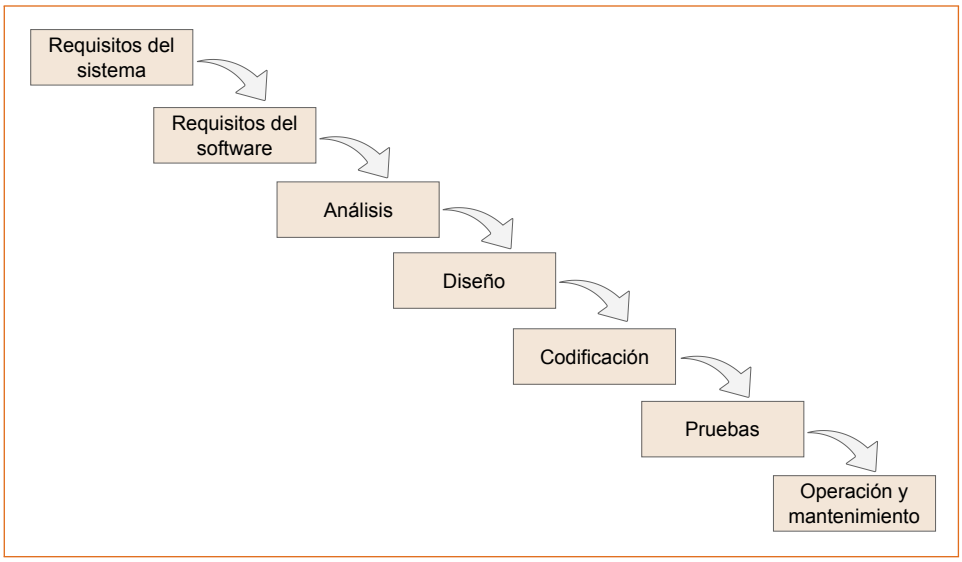


FIGURA 2.1: *Modelo de ciclo de vida en cascada*

Este modelo funciona bien en proyectos estables, normalmente proyectos pequeños en los que los requisitos están bien establecidos y no son cambiantes, puesto que es un modelo simple y bien organizado donde las fases no se superponen. Para la mayoría de los proyectos software este es, sin embargo, un modelo muy rígido. Así, en la práctica, es muy difícil conseguir que una fase del ciclo de vida de un producto software se termine completamente antes de moverse a las siguientes fases. Por ejemplo, es bastante improbable que un cliente establezca todos los requisitos del sistema software al inicio del proyecto, o que éstos no cambien a medida que avanza el proyecto. Además, los resultados y mejoras no son visibles progresivamente, es decir, el producto se ve únicamente cuando ya está finalizado, lo cual provoca una gran inseguridad por parte del cliente, el cual normalmente quiere ver los avances en el producto a lo largo del tiempo y, por tanto, hace difícil detectar discrepancias con las expectativas.

El modelo en cascada acepta múltiples modificaciones, como son la inclusión de fases extra, iteraciones entre fases, o la construcción de prototipos intermedios que permiten al cliente examinar y validar diferentes aspectos a implementar en el sistema en construcción. De esta forma, a pesar de los inconvenientes que tiene, el modelo en cascada sigue siendo muy popular en la industria [236, 276]. Aún así, este modelo es demasiado rígido, por lo que han aparecido otras alternativas.

El *modelo en V* [297] se desarrolló para poner solución a algunos de los problemas mencionados del enfoque en cascada tradicional. En el modelo en cascada los defectos se encontraban demasiado tarde en el ciclo de vida, ya que las pruebas no se realizaban hasta el final del proyecto. Así, el modelo en V enfatiza las tareas de pruebas, que en este modelo se integran en cada fase del ciclo de vida, elaborando planes de pruebas para cada una de las fases de definición, esto es, requisitos, análisis, diseño y codificación (ver figura 2.2). De todas formas, el modelo en V, al igual que el modelo en cascada, sigue siendo demasiado secuencial y, además, no proporciona caminos claros para los problemas encontrados durante las fases de pruebas que tienen su origen en una fase diferente.

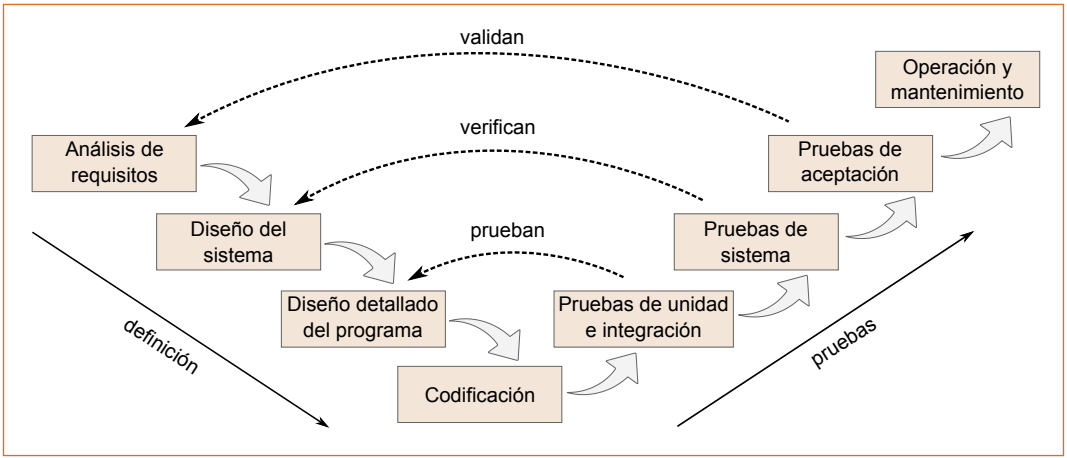


FIGURA 2.2: *Modelo de ciclo de vida en V*

Uno de los enfoques que intenta dar más flexibilidad es el *modelo de prototipos* [127]. Este tipo de modelos se basa en la construcción rápida de prototipos con la finalidad de que ciertos aspectos del software sean visibles al cliente o al usuario final (ver figura 2.3). Este modelo, que se integra dentro del modelo en cascada, reduce el riesgo de construir productos que no satisfagan las necesidades de los clientes. Sin embargo, como contrapartida, existe riesgo de que el prototipo sea asimilado por el cliente como el producto final en construcción, y esto podría causar consecuencias negativas si se intenta usar este prototipo para construir el sistema final cuando no fue pensado para ello.

Basándose en esta última idea, surgen los *modelos evolutivos*, los cuales se basan

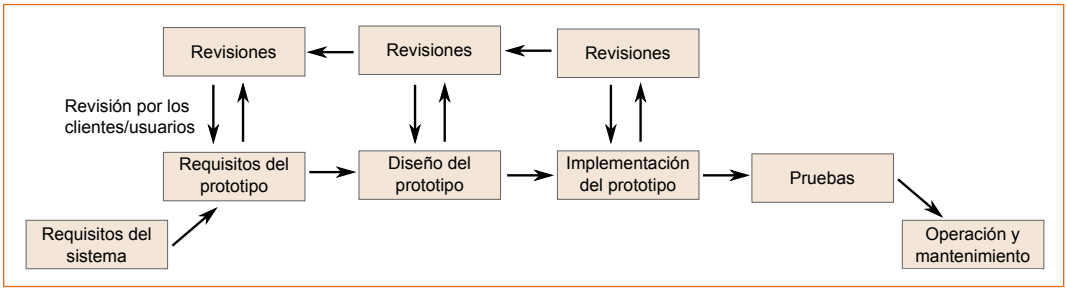


FIGURA 2.3: *Modelo de ciclo de vida de prototipos*

en la entrega periódica de versiones limitadas del producto que cumplan una serie de requisitos. Esto permite lanzar una versión de un producto al mercado mientras se están desarrollando nuevas versiones del mismo. Así, el *modelo de prototipos evolutivos*, el *modelo incremental*, el *modelo iterativo* y el *modelo en espiral* son ejemplos de modelos evolutivos.

El *modelo de prototipos evolutivos* [127] consiste en usar los prototipos de tal forma que, en vez de desecharlos, se refinan iterativamente para finalmente obtener el software objetivo. En el *modelo incremental* [237], por su parte, se divide el sistema en diferentes funcionalidades o subsistemas, y se repiten las fases de requisitos, diseño, implementación y pruebas, propias del modelo en cascada, para cada uno de ellos, entregando versiones parciales del software hasta conseguir el producto final (ver figura 2.4). Aunque este modelo aporta más flexibilidad, a veces es complejo definir el núcleo operativo para poder lograr el primer incremento. Por contrapartida, el *modelo iterativo* [237], difiere en la forma de obtener el producto final, puesto que, en este caso, se construye una primera versión rudimentaria y sin optimizar del sistema completo (probablemente con partes implementadas como prototipos que se desecharán), y se irá mejorando con cada evolución hasta que la calidad del producto encaje con los términos acordados con el cliente (ver figura 2.4). Basándose en el modelo iterativo, el *modelo en espiral* [121] trata el proyecto en ciclos, y cada uno de ellos incorpora objetivos de calidad y gestión de riesgos, así como una revisión que incluye todo el ciclo anterior y el plan para el siguiente (ver figura 2.6).

Los modelos evolutivos mejoraron a los modelos puramente secuenciales produciendo mejores resultados en la producción de software, entre otros aspectos porque transcurre menos tiempo entre que las necesidades del cliente se establecen hasta que dichas necesidades se incorporan como funcionalidades en el software. Sin embargo, todavía era necesario convertir este proceso en algo más dinámico, evitando algunos de los costes causados por la excesiva documentación y otras tareas administrativas, reduciendo, de esta forma, el tiempo en el que los clientes pueden tener acceso al software con las necesidades requeridas [122].

El primer enfoque que nace con la idea de entregar sistemas de forma rápida es

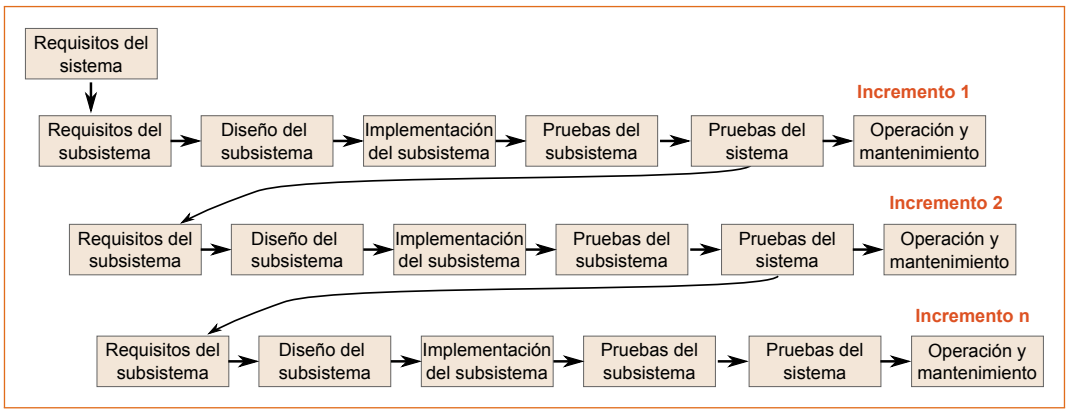


FIGURA 2.4: *Modelo de ciclo de vida evolutivo incremental*

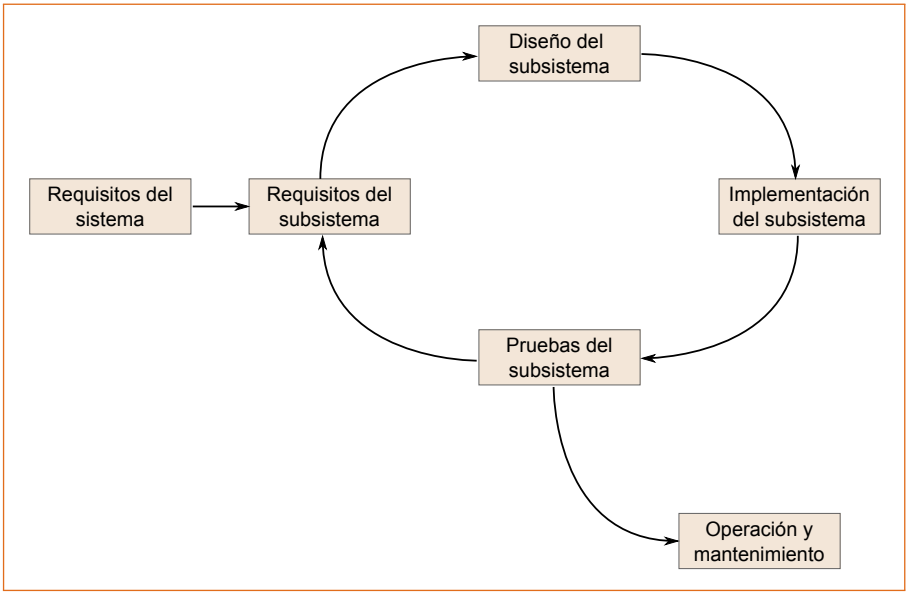


FIGURA 2.5: *Modelo de ciclo de vida evolutivo iterativo*

el *desarrollo rápido de aplicaciones* (RAD) [258]. La principal filosofía de este enfoque, desarrollado durante los años 80 y finalmente formalizado en 1991 por James Martin, es perseguir ciclos de vida cortos para conseguir una mayor velocidad de desarrollo. Este modelo se basa en el desarrollo iterativo y la construcción de prototipos, añade de forma explícita actividades de comunicación y de planificación como parte del ciclo de vida, y fomenta la existencia de diferentes grupos de desarrollo que trabajan en paralelo en diferentes partes del sistema (ver figura 2.7).

Siguiendo los pasos del modelo de desarrollo rápido RAD, a mediados de los años 90 surgen una serie de metodologías como Scrum [321], Crystal Clear [148], XP [108, 109], ASD [209], FDD [146, 288] o DSDM [331] que han ido evolu-

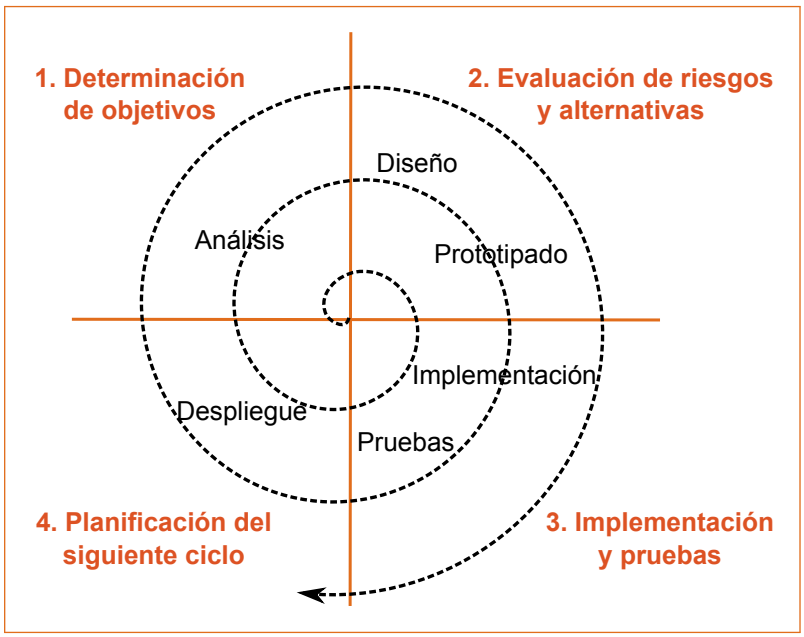


FIGURA 2.6: *Modelo de ciclo de vida evolutivo en espiral*

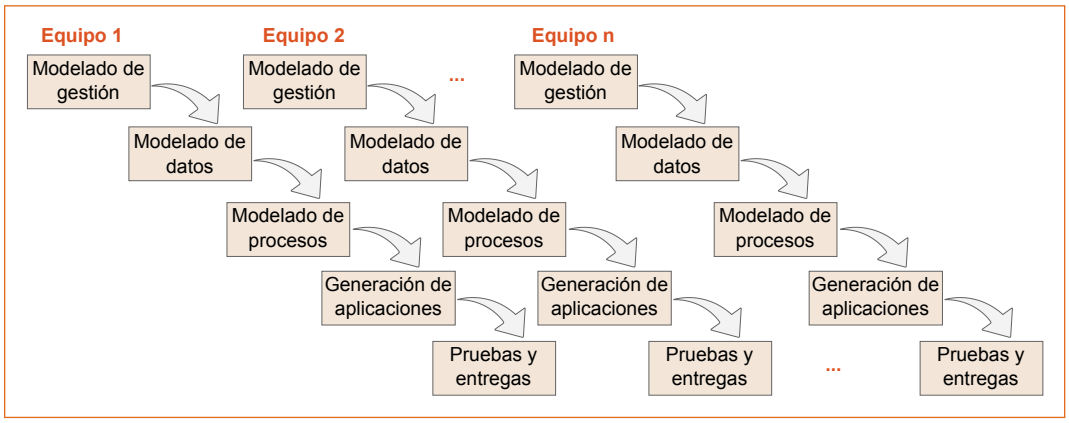


FIGURA 2.7: *Desarrollo rápido de aplicaciones*

cionando hasta que en febrero del 2001 se publicó el *Manifiesto Ágil* (*"Manifesto for Agile Software Development"*), a partir del cual se hace referencia a todas ellas como *metodologías ágiles* [150]. El *Manifiesto Ágil* es ampliamente considerado como la definición canónica del desarrollo ágil.

Las metodologías ágiles fomentan el trabajo en equipo, la organización y responsabilidad propia, y las capacidades de intercomunicación de las personas y sus conocimientos técnicos. La composición del equipo en un proyecto ágil es normalmente multidisciplinar y de organización propia sin consideración de la posible jerarquía corporativa, o los roles corporativos de los miembros de los equipos. La

filosofía común detrás de esta tendencia innovadora es la verdadera intención de invertir todo el tiempo y recursos en producir realmente buen software. Los métodos ágiles suelen incluir comunicación cara a cara rutinaria, diaria e informal entre los miembros del equipo, enfatizando la comunicación verbal sobre los documentos escritos. Así, el esfuerzo excesivo dedicado a la gestión, y el trabajo de producir multitud de informes y documentación, muchas veces desechado, se sustituye por una colaboración más activa con el cliente y dentro de los equipos de trabajo, centrándose en responder a los cambios, en lugar de en la creación y seguimiento de un plan de principio a fin, es decir, enfatizando más la adaptabilidad que la previsibilidad. Por tanto, el desarrollo ágil elige llevar a cabo un proyecto en incrementos pequeños y con una planificación mínima, en vez de planificaciones a largo plazo.

Finalmente, cabe destacar el *proceso de desarrollo de software unificado* [217], descrito en 1999 por Ivar Jacobson, Grady Booch, y James Rumbaugh como un marco de trabajo de software, que puede especializarse para una gran variedad de sistemas, y que, como características principales, destaca que está dirigido por casos de uso, centrado en la arquitectura, y sigue una filosofía iterativa e incremental. Este modelo describe el proceso de desarrollo de software como una combinación de tres puntos de vista o perspectivas diferentes: una perspectiva dinámica que muestra las fases del modelo en el tiempo (inicio, elaboración, construcción, y transición; cada una de ellas dividida en una serie de iteraciones que representan incrementos, como se muestra en la figura 2.8), una perspectiva estática que muestra las propiedades de cada una de las actividades en proceso, y una perspectiva práctica que sugiere buenas prácticas que deben aplicarse durante el proceso.

2.4. Las pruebas del software

Los programadores son (al menos de momento) seres humanos, y los seres humanos son propensos a cometer errores en las diferentes facetas de la vida. Evidentemente, el desarrollo de software no es una excepción. Los errores cometidos en las actividades de desarrollo de software (requisitos, diseño o codificación) suelen introducir *defectos* en el software. A su vez, estos defectos provocan que se produzcan *fallos*, los cuales se definen como un comportamiento no deseado del software [3].

Las actividades de pruebas son una parte integral y necesaria en el desarrollo de software, y se realizan para evaluar y mejorar la calidad de un producto software mediante la identificación de defectos en el mismo. Como se comentó anteriormente, las primeras metodologías de desarrollo de software [132] las retrasaban a las etapas finales de los proyectos de desarrollo, sin embargo, la tendencia actual en las metodologías modernas es realizar las pruebas lo antes posible. De esta forma, uno de los principales beneficios es detectar posibles defectos cuanto antes, puesto que, cuanto antes se detecten, más efectivas y fáciles serán las correcciones a realizar en el software [124, 215, 343].

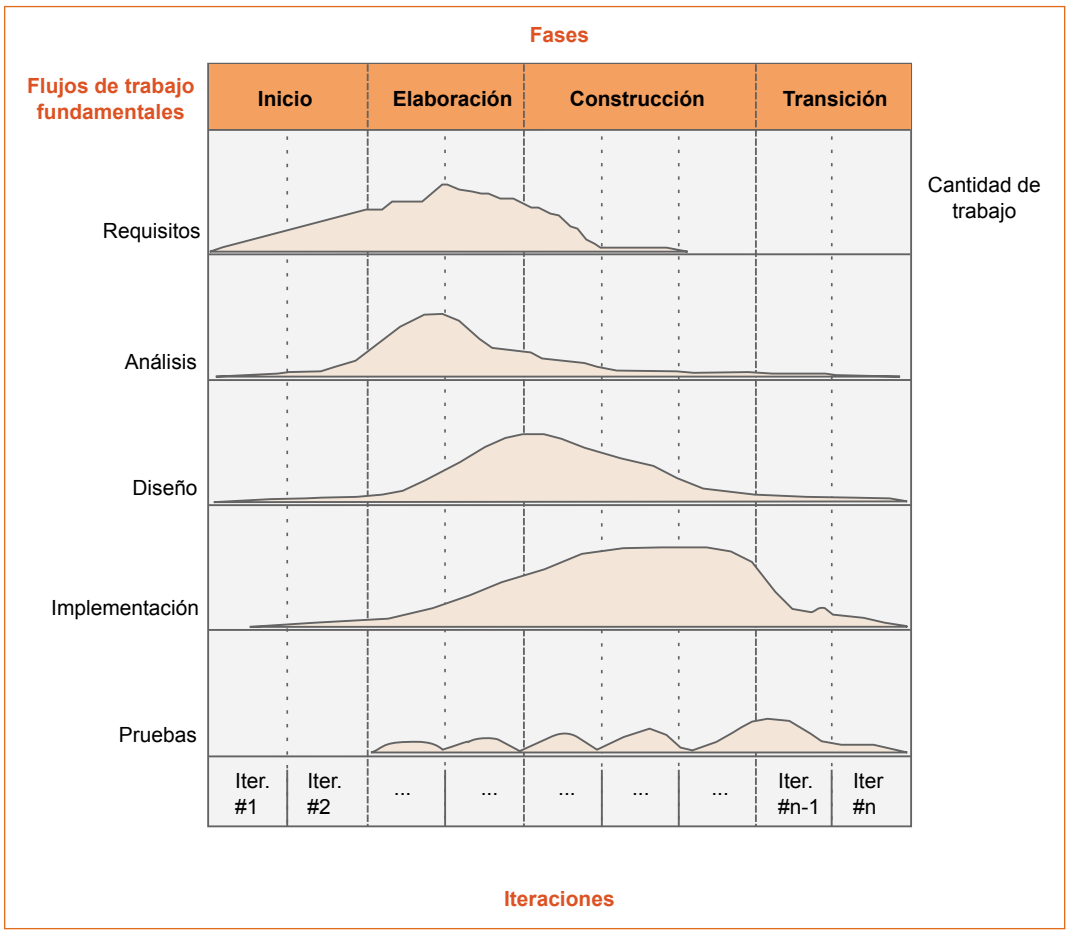


FIGURA 2.8: *Proceso de desarrollo de software unificado*

Por ejemplo, el proceso de desarrollo unificado de software [217] obliga a realizar una fase de pruebas después de cada iteración en vez de concentrar todo el esfuerzo relacionado con las pruebas en las fases finales del proyecto, como ocurre en las metodologías clásicas en cascada. Las metodologías ágiles [108, 309], por su parte, ponen incluso más énfasis en las pruebas. La idea principal es que tener buenos conjuntos de pruebas facilitan los cambios rápidos en el software preservando la calidad del mismo, y permiten a los equipos de desarrollo de software adaptar rápidamente el código a los cambios de requisitos.

Por otro lado, la visión que se tiene de las pruebas del software ha evolucionado desde los orígenes del software hasta la actualidad. Así, en 1988, Gelperin y Hetzel definieron esta evolución [194] en cinco fases diferentes:

- **Depuración (antes de 1956):** donde no había una diferencia clara entre los conceptos de depuración y pruebas.

- **Demostración (1957-1978):** en este período se establece una diferencia entre depuración y pruebas, siendo el objetivo de estas últimas demostrar que el software se comporta según sus especificaciones y requisitos.
- **Destrucción (1979-1982):** donde intentar demostrar que el software funciona como se espera cambia hacia un enfoque en el que el objetivo principal es forzar el mal funcionamiento del software para poder encontrar defectos en el mismo.
- **Evaluación (1983-1987):** cuya intención es la de proporcionar una evaluación del producto software y una medición de su calidad a lo largo del ciclo de vida del mismo, no sólo de su implementación, sino también de los requisitos y diseño.
- **Prevención (a partir de 1988):** que se basa en tener en cuenta las actividades de pruebas lo antes posible y, de esta forma, poder revelar defectos antes de pasar a la siguiente etapa de desarrollo.

Por tanto, las pruebas del software ya no se ven como una actividad que comienza justo después de que termine la fase de codificación, como ocurre en el modelo de ciclo de vida en cascada. En la actualidad, las pruebas del software se ven como una actividad que debe abarcar todo el proceso de desarrollo y mantenimiento del software, y son, en sí mismas, una parte importante en la construcción del producto software. De esta forma, se considera que uno de los caminos hacia la calidad del software es la prevención, puesto que es mucho mejor evitar los problemas que corregirlos.

Aunque como se comentará en las siguientes secciones, existen diferentes tipos de actividades que encajan dentro de las pruebas del software, como son las revisiones, inspecciones o verificación formal, entre otras, este trabajo utilizará el término *pruebas del software* para referirse a lo que comúnmente se corresponde con el término inglés *software testing*. Así, con este tipo de pruebas se realiza una verificación *dinámica* del comportamiento de un programa usando un conjunto *finito* de casos de prueba, *seleccionados* de forma adecuada de entre las posibles ejecuciones normalmente infinitas del dominio, evaluadas contra el comportamiento *esperado* [79]. Esta definición contiene cuatro palabras clave que caracterizan a este tipo de pruebas del software:

- *Dinámica*: al contrario que en las técnicas estáticas (revisiones, inspecciones, análisis de código estático, etc.), este tipo de pruebas requieren que se ejecute el programa a probar con unos valores de entrada, puesto que el sistema puede reaccionar de manera diferente ante los mismos valores de entrada.
- *Finito*: las pruebas exhaustivas no son prácticas en la mayoría de los programas reales, por el enorme tiempo requerido para su ejecución. Incluso para programas pequeños suele haber un gran número de entradas permitidas para

cada operación, junto con entradas inválidas o inesperadas, lo cual provoca que el número posible de secuencias de operaciones sea muy grande o, muchas veces, infinito. Por tanto, se debe llegar a un compromiso entre los recursos disponibles y el número de casos de prueba ejecutados.

- *Seleccionado*: debido a que el conjunto de casos de prueba posibles es muy grande o infinito, y sólo es posible ejecutar las pruebas con una pequeña representación de las mismas, se debe realizar una selección adecuada de qué casos de prueba se usarán para ejecutar las pruebas. Evidentemente, seleccionar los casos de prueba que tengan más probabilidad de causar fallos en el sistema, y eliminar casos de prueba que vayan a producir el mismo comportamiento que otros casos de prueba ya incluidos, no suele ser una tarea fácil. De hecho, muchas veces, la inclusión de unos casos de prueba u otros dependen de la experiencia de la persona que está realizando las pruebas. Además, se debe tener en cuenta que diferentes criterios de selección suelen producir diferentes grados de efectividad. Muchas de las técnicas de pruebas existentes se diferencian únicamente en cómo se seleccionan los casos de prueba.

- *Esperado*: después de cada ejecución se debe decidir si el comportamiento observado en el sistema es o no es aceptable, es decir, si se ha producido un fallo o no. El comportamiento observado se puede comparar con las expectativas del usuario, contra una especificación, o bien contra comportamiento esperado de requisitos implícitos.

Con esta definición se puede concluir que realizando las pruebas del software de esta forma no es posible demostrar que el software que prueban está libre de defectos, o que se comportará como se espera en cada circunstancia posible. De hecho, las pruebas del software, desde este punto de vista, sólo pueden mostrar la presencia de errores, nunca su ausencia [154]. A pesar de esto, este tipo de pruebas ayudan a detectar multitud de defectos que pasarían desapercibidos si éstas no se llevasen a cabo y, por esto, son una actividad muy importante dentro del desarrollo de software.

2.4.1. Verificación y validación

Verificación y *validación* son dos términos ampliamente usados en el campo de las pruebas del software. Ambos están relacionados con la calidad del software, y se refieren al conjunto de procesos de comprobación y análisis que determinan si el software que se desarrolla es acorde a su especificación y cumple las necesidades de los clientes. Estos dos procesos se llevan a cabo a lo largo del ciclo de vida de un producto software, comenzando con las revisiones de la especificación de requisitos, continuando con revisiones del diseño, inspecciones de código y las pruebas del software, entre otras actividades.

Sin embargo, existen diferencias entre estos dos términos. Así, mientras que la verificación establece la correspondencia entre un producto y su especificación (por ejemplo, si un documento de análisis resultante de una fase de análisis cumple con la especificación establecida, o si el código de un programa ha sido implementado acorde con la especificación); la validación establece si un producto software es adecuado para su uso previsto y en el entorno operativo previsto (por ejemplo, si un producto software cumple con las expectativas del cliente). De manera informal, se puede definir el proceso de verificación mediante la pregunta *¿se está construyendo el producto correctamente?*, mientras que la pregunta correspondiente para el proceso de validación sería *¿se está construyendo el producto correcto?* [120]. Así, la verificación ayuda a determinar si un producto tiene una alta calidad, pero no a comprobar si dicho producto es realmente útil.

Existen algunas definiciones de estos términos que pueden conllevar un uso diferente de los mismos en función del ámbito en el que se utilicen. Así, por ejemplo, dentro del modelo CMMI [80, 143, 326], el cual contiene un conjunto de prácticas que ayudan a las organizaciones a mejorar el proceso de desarrollo de software dentro de las mismas, las actividades de validación evalúan, durante o al final del proceso de desarrollo, y con el cliente siempre involucrado, si un producto (requisitos, diseños, programas, interfaces de usuario, manuales de usuario, materiales de formación, etc.) cumple con sus expectativas, es decir, lo que normalmente se conocen como *pruebas de aceptación*. Por ello, en muchas ocasiones, el término validación se asocia a *validación por el usuario*.

De la misma forma, en algunos contextos, el término *pruebas* se relaciona con verificación, y éste, a su vez, se asocia únicamente con *verificación formal*, que consiste en encontrar una demostración formal de la corrección de un programa [82]. Cabe destacar que en este trabajo, el término *pruebas del software* se refiere a la definición mostrada anteriormente, es decir, las técnicas que se encargan de comprobar el funcionamiento esperado del software a través de ejecuciones del mismo, usando para ello un conjunto finito de casos de prueba seleccionado de forma adecuada. Así, según el caso específico en el que se aplique, las pruebas del software pueden encajar dentro de las actividades de verificación o dentro de las actividades de validación.

2.4.2. Pruebas estáticas, dinámicas y simbólicas

Las técnicas de pruebas *estáticas* (o análisis estático) son aquellas que no requieren la ejecución del programa a probar, sino que se basan en examinar la documentación del proyecto, o el código fuente, pero sin ejecutarlo. Este tipo de técnicas se corresponden con actividades propias de verificación. Las técnicas de verificación estática más usadas son las *inspecciones de software* y las *revisiones*, destacando las *revisiones de pares*.

Las inspecciones y las revisiones [328] analizan y comprueban las diferentes partes del software, desde el código fuente hasta la documentación asociada, como son la especificación de requisitos o los modelos de diseño, e incluso las propias pruebas del sistema. Estas técnicas pueden llevarse a cabo sobre versiones incompletas de un sistema para detectar errores u omisiones antes de que esté disponible una versión ejecutable del mismo, y así observar si los estándares de calidad se han seguido correctamente. Durante una revisión, una persona o un grupo de personas examina el código fuente o la documentación asociada buscando problemas potenciales y otros atributos relacionados con la calidad, como son el cumplimiento de estándares, portabilidad y mantenibilidad. Así, en el código fuente, se podrían buscar ineficiencias, algoritmos inapropiados, variables sin inicializar o liberar, o un estilo de programación inadecuado, entre otros aspectos. Cabe destacar que el propósito de las revisiones e inspecciones es mejorar la calidad del software [257], no evaluar el rendimiento del equipo de desarrollo de software.

Otra de las técnicas de verificación estática existentes son las técnicas de verificación formal, también conocidas como *métodos formales* [82]. Estas técnicas de pruebas intentan demostrar que el comportamiento de un programa es correcto con respecto a una especificación o modelo formal no ambiguo del software. Para la construcción de este modelo, se deben traducir los requisitos de usuario, expresados en lenguaje natural, diagramas, tablas u otros mecanismos, en una notación matemática que defina formalmente la semántica del sistema. Por ejemplo, se podrían usar máquinas de estados finitos, redes de Petri, álgebra de procesos, semántica operacional, semántica denotacional, lógica de Hoare, etc. Existen dos enfoques principales dentro de la verificación formal:

- **La comprobación de modelos (*model checking*)**: que se basan en la construcción de un modelo finito del sistema, usado para comprobar si dicho modelo cumple o no una determinada propiedad. Esta comprobación se realiza como una exploración sistemática y exhaustiva del espacio de todos los estados del modelo, la cual está garantizada que termine, puesto que el modelo es finito. Uno de los retos de estas técnicas es la creación de algoritmos y estructuras de datos que permitan manejar búsquedas a través de una gran cantidad de estados.

- **La demostración de teoremas (*theorem proving*)**: en las que tanto el sistema como las propiedades a comprobar se expresan como fórmulas en algún tipo de lógica matemática. Esta lógica está dada por un sistema formal, el cual define un conjunto de axiomas y un conjunto de reglas de inferencia. Por tanto, se trata de encontrar una prueba formal de una propiedad a partir de un conjunto de axiomas del sistema.

Aunque las técnicas de verificación estática sí suelen ser usadas por compañías dedicadas al hardware, no suele ser habitual en compañías de desarrollo de software, salvo en algunos casos específicos para verificar el diseño y comportamiento de

sistemas software críticos para la seguridad [145]. Esto se debe a que usar este tipo de técnicas no es trivial, y suele ser necesario que éstas sean llevadas a cabo por expertos.

Por otro lado, las técnicas de pruebas *dinámicas* son aquellas que requieren la ejecución del programa a probar. Así, estas técnicas consisten en ejecutar el software con unos valores de entrada y observar si el comportamiento obtenido es el esperado. En este caso, normalmente, si el software se comporta de la manera esperada para unos valores de entrada seccionados, dicho software se asumirá que es correcto. Las técnicas de pruebas propuestas en este trabajo encajan dentro de esta categoría.

Finalmente, cabe destacar la existencia de una aproximación intermedia entre la verificación formal y las pruebas del software dinámicas, conocida como *ejecución simbólica* [230]. En este tipo de técnicas, en vez de ejecutar el programa a probar utilizando un conjunto de valores de entrada concretos, el programa se ejecuta "simbólicamente" para un conjunto de "clases" de entrada, las cuales vienen determinadas por las instrucciones de control de flujo que involucren las propias variables de entrada. Así, los valores de las variables del programa se representan como expresiones simbólicas que usan los valores simbólicos usados como valores de entrada. De esta forma, el resultado de cada ejecución simbólica puede ser equivalente a la ejecución de un conjunto de numerosos casos de prueba concretos. Estos resultados se pueden comprobar de manera formal o informal contra los resultados esperados. Cabe destacar que en los últimos años ha aumentado el interés por este tipo de técnicas debido a su capacidad para generar conjuntos de casos de pruebas con una cobertura alta, así como encontrar defectos en aplicaciones software complejas [130], siendo incluso utilizadas para la realización de las pruebas de sistemas software críticos [149].

2.4.3. Niveles de pruebas del software

Las pruebas del software que se realizan a lo largo del ciclo de desarrollo y mantenimiento de un producto software se pueden llevar a cabo en diferentes niveles en función del objetivo a probar, esto es, una única unidad de software, un componente, un grupo de componentes o el sistema completo. En función de estos criterios pueden distinguirse diferentes niveles de pruebas [83], los cuales se corresponden con los especificados en el modelo de ciclo de vida en V:

- **Las pruebas de unidad:** comprueban el funcionamiento de manera aislada de piezas de software que pueden ser probadas de forma separada. Dependiendo del contexto, estas piezas de software pueden ser una función, un método, una clase, un módulo o, en general, una "unidad" individual dentro del diseño de un sistema software que puede ser probada de forma independiente. Estas pruebas son realizadas siempre por los programadores.

- **Las pruebas de componente:** un componente software es una composición de unidades de software, con interfaces bien definidas y especificadas, que normalmente puede ser instalado de forma independiente, así como ser usado por terceros u otros componentes en sistemas software [340]. Las pruebas de componentes están diseñadas para evaluar estos componentes individuales de manera aislada, incluyendo las interfaces que se usarán para la interacción entre ellos, y las estructuras de datos asociadas en tal comunicación. Estas pruebas pueden ser realizadas por programadores o por un equipo de probadores de software.
- **Las pruebas de integración:** además de probar los componentes que forman parte de un sistema de manera independiente, es necesario combinarlos para probarlos como un grupo, y así comprobar que las interacciones entre ellos se han implementado de la manera adecuada. Este es el objetivo principal de las pruebas de integración, las cuales se deben llevar a cabo asumiendo que los componentes integrados funcionan de la manera esperada de forma separada. Suele ser el propio equipo de desarrollo el que lleva a cabo este tipo de pruebas.
- **Las pruebas de sistema:** estas pruebas se encargan de determinar si el sistema integrado cumple sus especificaciones. Asume que tanto los componentes individuales como las interacciones entre ellos funcionan de la manera esperada, y tratan al sistema como un todo. Este tipo de pruebas son normalmente realizadas por un equipo de probadores de software, aunque también podrían ser los propios programadores quienes las realicen.
- **Las pruebas de aceptación:** estas pruebas se encargan de comprobar si el software completado cumple las necesidades por las que dicho software ha sido creado. Estas actividades de pruebas pueden o no involucrar a los desarrolladores del sistema, pero siempre involucrarán al cliente o usuarios.

Este trabajo se centra fundamentalmente en las pruebas de unidad (capítulo 5), las pruebas de componente (capítulos 6 y 7) y las pruebas de integración (capítulo 8); y postula con menos profundidad cómo utilizar las técnicas propuestas, es decir, las pruebas basadas en propiedades, en las pruebas de sistema (capítulo 9).

2.4.4. Técnicas de pruebas del software

Existen numerosas técnicas de pruebas que pueden ser usadas para la realización de las pruebas del software [111, 112, 222, 225]. En las siguientes secciones, se describirán algunas de las principales, agrupadas según diferentes criterios [79].

2.4.4.1. Técnicas de pruebas no estructuradas

Este tipo de técnicas de pruebas se basan en la intuición y experiencia de la persona que esté realizando las pruebas del software para generar los casos de prueba

que se usarán en la ejecución de las mismas. Se pueden distinguir dos variantes:

- **Pruebas *ad hoc*:** en este enfoque los casos de prueba se derivan confiando en la habilidad, intuición y experiencia de quien esté realizando las pruebas con programas similares. Este tipo de técnicas de pruebas pueden ser útiles para la identificación de casos de prueba especiales, que no son fáciles de generar a través de otras técnicas formalizadas, o son difíciles de reproducir.

- **Pruebas exploratorias:** en las pruebas exploratorias, el aprendizaje, el diseño y la ejecución de las pruebas se realizan de forma simultánea. De esta forma, las pruebas no se definen por adelantado, sino que son diseñadas dinámicamente mientras que se prueba el sistema. Así, mientras que se está probando el software, la persona que realiza las pruebas también aprende cómo funciona dicho sistema y cómo se usa. Esta información, junto con su experiencia y creatividad, es la que se utiliza para generar nuevos casos de prueba a ejecutar.

  Por tanto, la efectividad de las pruebas exploratorias se basa en la habilidad y el conocimiento de las personas que diseñan los casos de prueba para generar aquellos casos que encuentren los defectos presentes. Este conocimiento se puede derivar de diversas fuentes: el comportamiento del software observado durante las pruebas, la familiaridad con dicho software y con la plataforma, los posibles tipos de fallos y defectos, el riesgo asociado al software en particular, etc.

  Aunque para algunos autores las pruebas exploratorias son equivalentes a las pruebas *ad hoc*, para otros la diferencia radica en que las pruebas *ad hoc* son totalmente desorganizadas, realizadas sin ningún tipo de criterio ni estructura, mientras que en las pruebas exploratorias se busca el aprendizaje del sistema a probar, obteniendo información de la ejecución de los casos de prueba para diseñar de forma dinámica otros nuevos casos de prueba.

2.4.4.2. Técnicas de pruebas basadas en la especificación

Estas técnicas usan la especificación del sistema a probar para generar, usando algún tipo de algoritmo, casos de prueba, sin conocer la estructura interna de dicho sistema a probar. Las técnicas más destacadas dentro de este tipo son las siguientes:

- **Partición en clases de equivalencia:** esta técnica consiste en dividir el dominio de entrada del programa a probar en un conjunto finito de *particiones* o *clases de equivalencia*, para las que se asume un comportamiento equivalente. Así, para cada condición de entrada se definen dos clases de equivalencia, los casos que cumplen esa condición y los casos que no lo hacen. De esta forma, se seleccionan casos de prueba en cada una de las clases identificadas. Utilizando esta técnica, un caso en una determinada clase es equivalente a

cualquier otro caso de prueba en dicha clase por lo que, si un caso de prueba en una clase produce un fallo, cualquier otro en la misma clase también lo produciría. De esta forma, se intenta reducir el número total de casos de prueba a utilizar. En algunas situaciones es necesario conocer las estructuras de datos y algoritmos usados en el programa para definir las particiones con el fin de diseñar los casos de prueba.

- **Análisis de valores límite:** esta técnica amplía las pruebas con clases de equivalencia, dirigiendo la selección de casos de prueba hacia los valores extremos de dichas clases de equivalencia, además de seleccionar casos con valores típicos dentro de cada clase. La justificación de esta estrategia deriva del hecho de que muchos de los fallos se producen en los límites de una determinada condición de entrada.

- **El método categoría-partición**: esta técnica, conocida como CPM [285], toma como base la técnica de particiones en clases de equivalencia para la generación sistemática de casos de prueba. En este método, la especificación se divide inicialmente en unidades funcionales que se pueden probar independientemente. Para cada unidad funcional, se identifican las características más destacables del dominio de entrada, las cuales se denominan *categorías*, y se divide cada una de ellas en clases de equivalencia. Los fundamentos de esta técnica son similares a la técnica de partición en clases de equivalencia, pero, en este caso, este método ofrece un enfoque sistematizado tanto para la partición del dominio de entrada como para la especificación y generación de casos de prueba. Así, las categorías se especifican en un lenguaje denominado *Test Specification Language* (TSL) a partir del cual se generan plantillas (*test frames*) para los casos de prueba individuales.

- **Grafos causa-efecto y tablas de decisión:** una de las mayores debilidades de las técnicas de particiones en clases de equivalencia es que no permiten combinar condiciones. Los grafos causa-efecto y las tablas de decisión, por el contrario, sí que lo permiten, y derivan, de forma sistemática, un conjunto efectivo de casos de prueba que puede revelar inconsistencias en una especificación.

  Para ello, la especificación del software, que podría estar formulada en lenguaje natural, debe ser transformada en un grafo lógico causa-efecto, que contiene operadores *booleanos* (identidad, conjunción, disyunción y negación). Como primer paso, la especificación inicial se descompone en unidades más manejables y, para cada una de ellas, se buscan *causas*, donde cada causa es un condición de entrada o una clase de equivalencia de condiciones de entrada, y *efectos*, que son las condiciones de salida o computaciones a ser realizadas por el sistema. Con esta información se construye un grafo con las relaciones entre causas y efectos, así como restricciones que describen combinaciones de causas y/o efectos imposibles. Finalmente, este grafo se convierte

en una tabla de decisión, que representa las relaciones entre las condiciones de entrada y el comportamiento o acciones que se producen como salida, a partir del cual se derivan los casos de prueba.

Las tablas de decisión también pueden generarse directamente sin necesidad de usar grafos causa-efecto. En cualquier caso, cabe destacar que estas técnicas son apropiadas cuando la lógica a probar está basada en decisiones, principalmente si dicha lógica puede expresarse en forma de reglas.

- **Pruebas de transición de estado:** estas técnicas de pruebas usan una máquina de estados que representa los estados, transiciones entre estados, eventos (entradas al sistema) y acciones (salidas del sistema ante los diferentes eventos) de un sistema software, para generar casos de prueba diseñados para ejecutar transiciones válidas e inválidas entre los diferentes estados. Esta técnica es útil cuando es posible representar el comportamiento de un sistema con un conjunto finito y manejable de estados, con transiciones entre ellos.

- **Pruebas por pares:** las pruebas por pares se basan en que la mayoría de los fallos están causados por interacciones de, como mucho, dos posibles factores o parámetros de entrada. Es por ello que los casos de prueba se diseñan para ejecutar todas las posibles combinaciones discretas de cada par de parámetros de entrada de un sistema. De esta forma, se intenta controlar la explosión combinatoria de factores de entrada para generar los casos de prueba. Para ello, se usa un criterio de cobertura mediante el cual se debe asegurar que cada factor y cada posible par de factores debe estar presente en, al menos, un caso de prueba, y cada factor y par de factores debe estar representado en el mismo porcentaje en todos los casos de prueba. Existen dos modelos básicos para las pruebas por pares:

  - **Todos los pares:** con esta técnica los casos de prueba son generados seleccionando de forma algorítmica y exhaustiva todas las posibles combinaciones de dos factores que se pueden producir.

  - **Técnica de matriz ortogonal:** este tipo de técnicas se conocen como OAT, y usan una matriz ortogonal [206] para generar los casos de prueba. Aunque la generación de casos de prueba con el modelo que combina todos los pares es normalmente más rápido, este modelo se basa en principios matemáticos que maximizan la cobertura con un número razonable de casos de prueba distribuyéndolos uniformemente en el dominio, reduciendo, por tanto, el tiempo de ejecución de las pruebas en comparación con el método que selecciona todos los pares.

- **Pruebas basadas en especificaciones formales:** a partir de una especificación escrita en un lenguaje "formal", derivada de los requisitos del software, que describe todos o algunos aspectos del sistema software a probar (normalmente propiedades funcionales), este tipo de técnicas generan casos de

prueba, proporcionando las salidas esperadas y, por tanto, las comprobaciones oportunas a realizar para evaluar el resultado de la ejecución de dichos casos de prueba. Las *pruebas basadas en modelos* [166, 208, 348], o el uso de especificaciones algebraicas [115] para derivar los casos de prueba son algunos de los métodos existentes en esta categoría.

- **Pruebas aleatorias:** a partir de la especificación del software a probar se generan, de manera aleatoria, los casos de prueba. Por ejemplo, las *pruebas basadas en propiedades* [182] utilizan una especificación en forma de propiedades a partir de la cual se generan de forma aleatoria casos de prueba que permiten comprobar si dichas propiedades se cumplen o no en una implementación concreta a probar.

2.4.4.3. Técnicas de pruebas basadas en el código

Estas técnicas se encargan de generar casos de prueba en función de un criterio de cobertura del código, que indicará cómo generar los casos de prueba, así como el criterio de parada, es decir, cuándo se debe parar de generar casos de prueba porque ya se ha alcanzado el nivel de cobertura del código esperado. Entre los criterios de cobertura existentes se encuentran:

- **Criterios basados en el flujo de control:** los casos de prueba se generan basándose en el conocimiento de la estructura de control del programa a probar. A su vez, existen diferentes criterios de cobertura [274] como son la cobertura de sentencias, la cobertura de ramas o decisiones, la cobertura de caminos, la cobertura de condiciones, la cobertura de condiciones/decisiones, o el criterio de cobertura de condición/decisión modificada (MC/DC) [142], entre otros.
- **Criterios basados en el flujo de datos:** los casos de prueba se generan basándose en el conocimiento de las operaciones que se realizan sobre las variables en el programa a probar [306]. La idea principal es cubrir caminos del programa a probar en los que aparezca una determinada variable o variables. Para ello, se atiende a diferentes criterios basados en la definición de una variable, su uso, y el camino desde que se define una variable hasta que se usa.

2.4.4.4. Técnicas de pruebas basadas en fallos

Este tipo de técnicas de pruebas se basan en la utilización de casos de prueba especialmente diseñados para revelar fallos probables o predefinidos. Destacan las siguientes técnicas:

- **Predicción de error (*error guessing*):** los casos de prueba se diseñan con la intención de "averiguar" qué defectos podrían estar presentes en el componente a probar, basándose únicamente en la experiencia de la persona que esté realizando las pruebas, por ejemplo, como resultado de los errores cometidos anteriormente o la historia de fallos descubiertos en proyectos anteriores. El

alcance de los casos de prueba suele ser tratar de encontrar aquellas situaciones que suelen causar fallos en el software, como son, por ejemplo, divisiones por cero, punteros nulos o el comportamiento ante parámetros inválidos.

- **Pruebas de mutación:** un mutante es una versión ligeramente modificada (en tiempo de compilación o en tiempo de ejecución) del programa a probar, que difiere en un pequeño cambio sintáctico del programa original. Cada caso de prueba se ejecuta tanto con el código original como con cada uno de los mutantes generados, de tal forma que si un caso de prueba funciona con el programa original, pero no con un mutante, identificando, por tanto, la diferencia entre ellos, dicho mutante se dice que ha sido "matado". Originalmente esta técnica ha sido concebida para evaluar la calidad de un conjunto de casos de prueba existentes, contando cuántos mutantes han "sobrevivido", y obteniendo, de esta forma, una medida de cobertura.

  No obstante, también puede ser considerada un criterio de pruebas en sí mismo. Así, o bien los casos de prueba se generan al azar hasta que suficientes mutantes hayan sido "matados"; o bien los casos de prueba se diseñan específicamente para "matar" mutantes "supervivientes". En este último caso, este tipo de pruebas también puede ser categorizadas como una técnica de pruebas basada en el código. En cualquier caso, para que la técnica sea eficaz, deben derivarse de forma automática un gran número de mutantes de una manera sistemática. Sin embargo, éste suele ser un proceso bastante costoso, sobre todo cuando se utiliza con aplicaciones grandes.

- **Pruebas de inyección de fallos:** la inyección de fallos es una técnica de pruebas que simula fallos en ciertas partes del código de un programa, con el objetivo de determinar si el sistema, con dicho fallo, se comporta de la manera adecuada al ser ejecutado. Esto es útil en aquellas aplicaciones en las que se deben realizar acciones muy complejas para provocar un fallo. En estos casos, los fallos ocurren con muy poca frecuencia y, por tanto, es difícil evaluar cuánto de tolerantes son estas aplicaciones a la ocurrencia de los mismos. De esta forma, insertar fallos deliberadamente en el sistema ayuda a la reproducción de los mismos de manera rápida.

  La diferencia de este tipo de pruebas con las pruebas de mutación es que, aunque ambas se basan en insertar fallos en el sistema a probar, el objetivo de las pruebas de mutación es medir la efectividad de un conjunto de casos de prueba, así como aumentar la cobertura del código de las pruebas a partir de este análisis. Por otro lado, las pruebas de inyección de fallos tienen como objetivo principal determinar si el sistema a probar maneja de forma adecuada diferentes tipos de fallos.

- ***Fuzzing*:** las técnicas de *fuzzing* [266, 339, 342] son un tipo de técnicas de inyección de fallos, que consisten en usar datos inválidos, inesperados y alea-

torios (llamados *fuzz*) como entrada a un sistema con el objetivo de forzar la ocurrencia de fallos. Hay dos aproximaciones principales para crear los casos de prueba: por mutaciones de muestras de datos existentes, o generando nuevos datos de entrada basados en modelos de la entrada del sistema. En cualquier caso, este tipo de técnica no intenta "adivinar" qué datos de entrada harán que la aplicación no funcione de la manera esperada, sino que se usan datos de entrada completamente aleatorios.

La idea principal consiste en observar las respuestas del sistema cuando se produce un fallo, revelando, bajo circunstancias anómalas controladas, cómo de "mal" se comporta el software, cuantificando así la influencia de eventos externos sobre el comportamiento del mismo. Así, muchas veces, la realización de este tipo de pruebas no comprueba que el software se comporta de la manera esperada ante situaciones "normales", sino que únicamente podrá comprobar que el software puede manejar excepciones de forma apropiada ante situaciones inesperadas. De esta forma, estas técnicas pueden ayudar a determinar la tolerancia de un sistema, es decir, si el sistema es capaz de producir resultados aceptables ante la presencia de entradas malintencionadas o corruptas. Normalmente, las técnicas de *fuzzing* se usan para detectar vulnerabilidades, problemas de seguridad u otros problemas potenciales como fugas de memoria, desbordamiento de *búfer*, inyección de código SQL, etc.

2.4.4.5. Técnicas de pruebas basadas en el uso

Las técnicas de pruebas basadas en el uso tratan de evaluar la fiabilidad del software cuando es utilizado de la misma forma que lo usan los usuarios finales en el entorno de producción, con la finalidad de que los defectos puedan aparecer antes de que sea usado por usuarios reales. La fiabilidad del software se define como la probabilidad de que un sistema software funcione correctamente sin que se produzca ningún fallo durante un intervalo de tiempo, bajo una serie de condiciones.

Un tipo de técnica de pruebas basada en el uso del sistema es la construcción de un *perfil operacional* [270–272]. Un perfil operacional es una representación cuantitativa de cómo será usado un sistema. Así, modela cómo los usuarios usan el sistema, para lo que se especifica la probabilidad de ocurrencia de las llamadas a cada operación del sistema, y la distribución de los parámetros de entrada de las mismas. Tal descripción del comportamiento de los usuarios puede ser usada para generar casos de prueba de manera estadística.

Por tanto, el objetivo de estas técnicas es reproducir el entorno operativo en el que deberá funcionar el programa, con escenarios de uso, para realizar las pruebas sobre el mismo. De esta forma, se trata de inferir, a partir de los resultados observados, la futura fiabilidad del software cuando esté en uso.

2.4.4.6. Técnicas de pruebas según la naturaleza de la aplicación

Las técnicas de pruebas explicadas anteriormente pueden ser aplicadas a cualquier tipo de software. Sin embargo, existen técnicas de pruebas más específicas para realizar las pruebas de forma más eficiente y eficaz para algunos tipos de aplicaciones. Por ejemplo, es posible encontrar técnicas de pruebas específicas para los siguientes tipos de aplicaciones:

- **Aplicaciones orientadas a objetos** [232], en las que se debe tratar con las características de este tipo de aplicaciones, como son la existencia de clases y objetos, abstracción, encapsulación, herencia o polimorfismo.
- **Interfaces gráficas de usuario** [302], ya sea aplicaciones de escritorio o interfaces web. En este caso, se debe evaluar si la interfaz funciona de la manera esperada, teniendo en cuenta los elementos gráficos que se muestran, y cómo éstos reaccionan ante las interacciones con los usuarios.
- **Aplicaciones web** [163], en las que se deben evaluar diferentes criterios como son la funcionalidad, usabilidad, interfaz, rendimiento, seguridad, o accesibilidad de las mismas.
- **Programas concurrentes** [134], los cuales suelen ser más difíciles de probar que los programas secuenciales, puesto que, al igual que ellos, las pruebas también también deben ser ejecutadas concurrentemente, causando, en muchos casos, que los fallos encontrados no puedan ser reproducidos de manera determinista.
- **Sistemas en tiempo real** [320], los cuales están sujetos a restricciones estrictas de tiempo, es decir, las operaciones deben producir una respuesta en un tiempo determinado.
- **Sistemas de seguridad críticos** [335], esto es, sistemas software cuyo fallo o mal funcionamiento puede perjudicar gravemente la vida, el medio ambiente o algún tipo de equipamiento.

En referencia a las técnicas de pruebas explicadas en este trabajo, cabe mencionar que éstas no se centran en problemas específicos de un tipo de software o paradigma, sino que se han desarrollado de manera genérica.

2.4.4.7. Técnicas de pruebas según la finalidad de las mismas

Las pruebas del software se llevan a cabo con la finalidad de realizar algún tipo de comprobación. Por un lado, los casos de prueba pueden ser diseñados para comprobar que las especificaciones funcionales se han implementado correctamente, pero también se pueden utilizar para comprobar otro tipo de aspectos más específicos, dando lugar a otros tipos de pruebas según su finalidad. Algunos ejemplos son los siguientes:

- **Las pruebas de instalación**, que comprueban que el software puede ser instalado en el entorno objetivo.
- **Las pruebas de compatibilidad**, que comprueban que el software sigue funcionando con otras aplicaciones, sistemas operativos o entornos diferentes al original.
- **Las pruebas con *alpha o beta testers***, cuyo objetivo es que un grupo representativo de usuarios utilice el sistema antes de realizar su paso a la etapa en producción.
- **Las pruebas de rendimiento**, como son las pruebas de carga, las pruebas de estrés o las pruebas de resistencia, que evalúan la capacidad del sistema y los tiempos de respuesta ante una carga determinada o más allá de los límites para los que ha sido diseñado. Para ello se utiliza, en muchos casos, una aproximación de pruebas basada en el uso.
- **Las pruebas de recuperación**, que comprueban el comportamiento del sistema cuando se produce algún tipo de "desastre".
- **Las pruebas de usabilidad**, que evalúan cómo de fácil es para los usuarios finales usar y aprender a usar el software.
- **Las pruebas de accesibilidad**, que comprueban que el software es accesible, por ejemplo, para personas con visión reducida, audición reducida o movilidad reducida, entre otros aspectos.
- **Las pruebas de seguridad**, que comprueban la seguridad del sistema, por ejemplo, para protegerlo contra el acceso de usuarios no autorizados, o el abuso de usuarios autorizados.
- **Las pruebas de regresión**, que comprueban que las modificaciones realizadas en el sistema no causan comportamientos no deseados en otras partes del software que antes de la realización de dichas modificaciones funcionaban de la manera esperada.

2.4.5. Criterios de realización de las pruebas del software

La puesta en práctica de las diferentes técnicas de pruebas descritas en la sección anterior puede realizarse según diferentes puntos de vista en función del problema concreto. Así, en esta sección se describen los enfoques principales que pueden seguirse a la hora de aplicar una técnica de pruebas concreta.

2.4.5.1. Pruebas deterministas y aleatorias

Los casos de prueba pueden ser seleccionados de una manera determinista o pueden ser elegidos de forma aleatoria siguiendo un modelo de distribución de probabilidad estadístico para las entradas del sistema. De las técnicas listadas en la sección

anterior, existen algunas en las que los casos de prueba son elegidos de forma aleatoria por la naturaleza de las mismas, como son las técnicas basadas en el uso, o las propias pruebas aleatorias basadas en especificaciones, entre las que se encuentran las pruebas basadas en propiedades. Por el contrario, otras técnicas, como pueden ser las basadas en el código, pueden ser usadas con ambas aproximaciones de generación de casos de prueba [345].

2.4.5.2. Pruebas de caja negra, de caja blanca y de caja gris

Cuando se realizan las pruebas del software, éstas se pueden llevar a cabo según una perspectiva de *caja negra*, una perspectiva de *caja blanca* o una aproximación mixta, de *caja gris*, en función de si se tiene en cuenta la estructura interna del sistema a probar para la realización y ejecución de las mismas [228].

Las *pruebas de caja negra* tratan al sistema a probar desde un punto de vista externo. Así, estas pruebas no asumen ningún tipo de suposición sobre la estructura interna del sistema a probar, sino que evalúan el comportamiento del sistema a través de sus entradas y salidas. Los casos de prueba son pares de entradas, válidas e inválidas, con sus correspondientes salidas esperadas. De esta forma, el comportamiento del sistema se evalúa ejecutando el sistema a probar con los datos de entrada correspondientes y comparando las salidas obtenidas con las salidas esperadas para dichas entradas. Por ejemplo, las técnicas de pruebas basadas en especificaciones se realizan normalmente desde una perspectiva de caja negra.

Por otro lado, las *pruebas de caja blanca* hacen uso de la estructura interna del sistema a probar para derivar los casos de prueba. Así, por ejemplo, las técnicas de pruebas basadas en el código, que usan algún tipo de criterio de cobertura para generar los casos de prueba, encajan dentro de esta categoría.

Por último, las *pruebas de caja gris* combinan las pruebas de caja blanca con las pruebas de caja negra, de tal forma que una pieza de software se prueba contra sus especificaciones, involucrando entradas y salidas esperadas, pero usando conocimiento de cómo funciona internamente el programa. Por ejemplo, las pruebas de integración suelen realizarse con una aproximación de caja gris, puesto que al igual que las pruebas de caja negra estimulan al sistema desde un punto de vista externo, invocando una serie de operaciones con unos datos de entrada y obteniendo las salidas correspondientes; pero, sin embargo, se necesita el conocimiento de qué interacciones internas se deben producir con otros componentes para poder comprobar que realmente dichas interacciones esperadas se han realizado de la manera deseada.

2.4.5.3. Pruebas positivas y pruebas negativas

Según la estrategia que se adopte en el diseño de los casos de prueba, en concreto, en función de si éstos se diseñan para comprobar el buen funcionamiento del

software, o bien para demostrar que éste no funciona de la manera esperada, las pruebas pueden ser clasificadas como *positivas* o *negativas*.

Por un lado, las *pruebas positivas* comprueban si un sistema software cumple su especificación cuando éste se ejecuta bajo condiciones "normales". De esta forma, en este tipo de pruebas se usan únicamente datos de entrada válidos (valores de entrada "positivos") para comprobar si el sistema se comporta de la manera esperada ante este tipo de entradas.

Por otro lado, las *pruebas negativas* intentan causar de forma deliberada que el sistema falle, con el fin de comprobar si éste se comporta de manera apropiada ante estos fallos, y es capaz de manejar situaciones anómalas satisfactoriamente. En este caso, los casos de prueba están formados por datos de entrada no válidos, es decir, valores no controlados para los que el sistema no ha sido diseñado, los cuales permitirán comprobar si el software se comporta de la manera adecuada ante este tipo de entradas "negativas".

Los objetivos principales de ambos enfoques son completamente diferentes. Así, mientras que las pruebas positivas intentan comprobar que el software cumple con sus requisitos y especificaciones, las pruebas negativas tratan de comprobar la estabilidad del software ante la presencia de datos incorrectos. No obstante, ambos enfoques son complementarios.

2.4.5.4. Pruebas funcionales y pruebas no funcionales

En función del tipo de requisitos que se prueben, las pruebas del software pueden clasificarse como *pruebas funcionales*, si prueban requisitos funcionales, o *pruebas no funcionales*, si su objetivo son las pruebas de requisitos no funcionales.

Los requisitos funcionales definen las funcionalidades que el software debe llevar a cabo. Así, este tipo de requisitos describen el comportamiento del sistema definiendo las entradas al mismo, la descripción de las operaciones y el flujo de datos, así como las salidas de dicho sistema. Por tanto, cualquier tipo de prueba que compruebe el comportamiento funcional del sistema será considerado una prueba funcional.

Por otro lado, los requisitos no funcionales, en vez de describir algún tipo de comportamiento específico, imponen restricciones sobre el diseño o la implementación del sistema, como pueden ser el rendimiento, la seguridad, la fiabilidad, etc. Por tanto, probar este tipo de restricciones se considera un tipo de prueba no funcional. Algunos ejemplos de pruebas no funcionales son las pruebas de usabilidad, las pruebas de accesibilidad, las pruebas de rendimiento, o las pruebas de seguridad, entre otras.

2.5. Las pruebas basadas en propiedades

Las pruebas basadas en propiedades (PBT) [182] son una aproximación de pruebas automáticas que se basa en el uso de propiedades o modelos para describir el comportamiento del sistema a probar. Esta especificación en forma de propiedades o modelos se usa para generar automáticamente casos de prueba (datos de entrada a funciones, secuencias de llamadas a operaciones u otras representaciones), y para evaluar dichos casos de prueba comprobando si la especificación, es decir, las propiedades o modelos definidos, se cumple en dicho sistema. Así, esta es una aproximación de alto nivel, que no se centra en la especificación de casos de prueba individuales, sino que usa una serie de propiedades o modelos que expresan en forma lógica cómo se debe comportar el sistema a probar.

Escribir propiedades en vez de casos de prueba concretos tiene como implicación que los programadores deben pensar menos en la implementación y más en los resultados esperados. Además, puesto que las propiedades son mas informativas y mantenibles que los casos de prueba concretos, y reflejan el comportamiento del sistema en alto nivel, éstas pueden ser usadas, en muchas ocasiones, como documentación del sistema que prueban.

En general, las herramientas de pruebas basadas en propiedades reciben como entrada el sistema a probar y una especificación, en forma de propiedades o modelos, que describe dicho sistema (figura 2.9). Con estas entradas, se genera o bien un resultado positivo, siempre y cuando la especificación de entrada se cumpla para un conjunto de casos de prueba generados automáticamente, o bien un contraejemplo que indica cómo la propiedad o el modelo ha fallado.

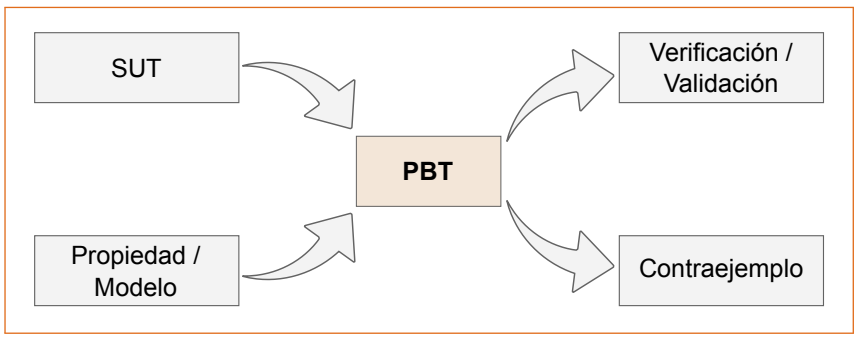


FIGURA 2.9: *Las pruebas basadas en propiedades*

Los casos de prueba producidos utilizando una aproximación basada en propiedades se obtienen mediante un proceso de generación, a partir de las propiedades o modelos definidos, basado en algún tipo de distribución aleatoria. Aunque esto podría parecer menos conveniente que una aproximación sistemática, existen estudios que indican que las pruebas aleatorias son más adecuadas en muchos casos [170, 203]. En cualquier caso, aunque se usen propiedades o modelos como especificación del sistema a probar, ésta no es una técnica de verificación formal,

puesto que no se comprueban todos los casos de prueba posibles, sino que únicamente se utilizan una serie de casos generados automáticamente para comprobar la especificación de entrada.

2.5.1. QuickCheck

QuickCheck es, sin duda, la herramienta de pruebas basadas en propiedades más conocida y exitosa en la actualidad. La herramienta QuickCheck fue originariamente desarrollada en Haskell en la universidad de Chalmers por Claessen y Hughes [144] en el año 2000 como herramienta de pruebas basadas en propiedades para programas escritos en Haskell. Actualmente, existen diferentes versiones de QuickCheck para diferentes lenguajes de programación: C++ [327], Java [224], ML [239] o Erlang [58, 256, 289], entre otros.

De entre todas las versiones disponibles, la versión comercial proporcionada por Quviq, conocida como Quviq QuickCheck [211], y que está implementada en Erlang, es la que posee más funcionalidades comparada con el resto de versiones. Esta es la versión que se usa en este trabajo como referencia en los ejemplos y casos de estudio utilizados. Además, cabe mencionar que en este trabajo se utilizará simplemente el término QuickCheck para referirse a la versión Quviq QuickCheck[1] .

La herramienta QuickCheck soporta la generación automática de casos de prueba así como la ejecución de los mismos. Para ello, QuickCheck proporciona un lenguaje de especificación de propiedades y modelos que permiten describir el comportamiento del sistema a probar. Con respecto a las propiedades, éstas se expresan en forma lógica. Por otro lado, el tipo de modelos soportado por QuickCheck son las máquinas de estados, las cuales están especialmente diseñadas para las pruebas de funciones relacionadas con efectos colaterales. Así, en las máquinas de estados se especifican una serie de funciones a probar, las cuales permitirán transicionar al sistema que se prueba entre estados al ser ejecutadas, con una serie de precondiciones y postcondiciones asociadas. Ejecutar este tipo de propiedades o modelos implica la generación aleatoria de casos de prueba a partir de los cuales se comprueban si dicha especificación se cumple o no para cada uno de ellos.

Esta herramienta comenzó a utilizarse en la industria, en concreto, en la compañía Ericsson, aunque su adopción fue lenta. Por otro lado, como resultado del proyecto europeo FP7 ProTest [162], se desarrollaron y mejoraron técnicas, herramientas y metodologías para la realización de pruebas basadas en propiedades, entre ellas, la propia herramienta QuickCheck. De esta forma, las pruebas basadas en propiedades y QuickCheck comenzaron a utilizarse con éxito en la industria para probar requisitos funcionales de estaciones base de radio (Ericsson), servidores de chat (Facebook y Tuenti), bases de datos de *back-end* (Basho – Riak), software embebido en coches (Volvo), etc. Además, en muchos casos de estudio se obtuvo una reducción del 90 %

[1] Aunque durante la realización de este trabajo han sido liberadas diferentes versiones de QuickCheck, cabe destacar que todos ejemplos mostrados funcionan con la versión actual v1.34.2.

en el tamaño del código de pruebas con respecto a código de pruebas existente, donde se especificaban los casos de prueba manualmente, a la vez que se incrementó la cobertura de las pruebas [89, 90].

Durante el proyecto ProTest, LambdaStream, uno de los socios industriales del proyecto, fue usada como caso de estudio para validar las herramientas y métodos desarrollados durante el mismo. En este estudio de investigación, de 22 meses de duración [277], las pruebas basadas en propiedades fueron introducidas en el proceso de desarrollo software de LambdaStream junto con el desarrollo dirigido por las pruebas [95, 107]. De esta forma, se ha podido evaluar la utilidad de un conjunto de herramientas de pruebas, como QuickCheck [144], Wrangler [242] o McErlang [192], en la industria, a través de una serie de entrevistas periódicas con los ingenieros de LambdaStream, y de la monitorización del uso de estas herramientas proporcionadas, obteniendo así datos de una forma cualitativa y cuantitativa.

Una de las principales conclusiones de este estudio es el gran esfuerzo que se necesitó en las primeras etapas del proceso para introducir las pruebas basadas en propiedades en la compañía, ya que fue mucho más grande de lo previsto, siendo la definición de propiedades, así como el depurado de errores encontrados durante las pruebas (tanto en el código a probar como en el código de pruebas), una gran barrera inicial. Soporte, cursos y documentación adicional fueron necesarios para una correcta adopción de las herramientas en LambdaStream. Sin embargo, una vez que se comenzó el uso regular de las herramientas por el equipo de desarrollo de LambdaStream por un período de tiempo, se obtuvieron conclusiones positivas acerca de las mismas. Una de las consecuencias positivas del uso de propiedades, en combinación con el desarrollo dirigido por las pruebas, es que éstas ayudan a pensar los requisitos y funcionalidades de antemano, lo cual permite escribir el código de una forma más clara y organizada, dando lugar a código de más calidad, más legible, modular, estructurado, mantenible y menos complejo. Además, el código resultante es más fácil de ser probado, y se tiende a escribir más pruebas y de mejor calidad. Por otro lado, se percibió como los defectos, en general, se solían detectar antes. Así, muchos defectos que antes no eran detectados hasta las etapas de integración de componentes, ahora se detectaban en etapas anteriores, aunque defectos complejos todavía permanecían ocultos hasta las etapas finales del desarrollo [277].

Cabe mencionar que en este proceso también se observó la necesidad de seguir desarrollando técnicas específicas para ser utilizadas en algunas situaciones. Por esta razón, tras la finalización del proyecto ProTest, se ha seguido trabajando en nuevas técnicas de pruebas basadas en propiedades más especializadas. En concreto, en el proyecto europeo FP7 PROWESS [4], se desarrollaron técnicas y herramientas de pruebas automáticas basadas en propiedades adaptadas para la realización de pruebas de servicios web de una forma eficiente y eficaz.

En cualquier caso, las ventajas observadas gracias al uso de esta técnica para la realización de las pruebas del software (también detalladas en otros trabajo que

describen experiencias con la herramienta QuickCheck como [87, 287, 291]), convierten a las pruebas basadas en propiedades en una técnica de pruebas potente que, además de requerir menos código fuente de pruebas que aquellas técnicas en las que se especifican los casos de prueba manualmente, permite incrementar la eficiencia y eficacia con respecto a éstas, mejorando así la productividad. Este hecho propicia que ésta sea la técnica de pruebas principal en la que se basa este trabajo para el desarrollo de aproximaciones especializadas para la realización de las pruebas del software en diferentes niveles.

3

CASO DE ESTUDIO

3.1. Introducción

Cada una de las metodologías y técnicas de pruebas desarrolladas en este trabajo serán ilustradas con un ejemplo real. En concreto, todos los ejemplos mostrados en este trabajo se corresponden con partes de un sistema interactivo de televisión digital llamado VoDKATV [35], el cual se encuentra actualmente en producción en diferentes instalaciones de clientes en todo el mundo (España, Alemania, Suiza y Emiratos Árabes). Así, a partir de estos ejemplos, se muestra cómo utilizar estas metodologías y técnicas de pruebas de una forma práctica. Además, el hecho de usar un sistema software real, en vez de ejemplos sintéticos, permite mostrar situaciones reales en las que estas metodologías y técnicas pueden ser aplicadas. De la misma forma, en algunas situaciones ha sido posible comparar el uso de estas nuevas metodologías y técnicas con las aproximaciones de pruebas que se estaban usando en dicho sistema VoDKATV.

El objetivo principal de este capítulo es describir el propio sistema VoDKATV. Para poder comprender el funcionamiento de este sistema, así como el entorno en el que se instala, se realiza, inicialmente, en la sección 3.2, una pequeña introducción al mundo de la televisión actual, haciendo referencia a la evolución que han sufrido los servicios de televisión en los últimos años. Posteriormente, la sección 3.3 explica cómo funcionan, de una manera genérica, los sistemas interactivos de televisión digital, como VoDKATV. Después de estas secciones introductorias, la sección 3.4 realiza una descripción del propio sistema VoDKATV, enumerando sus características principales, así como los entornos principales donde puede y ha sido instalado este sistema. A continuación, la sección 3.5 describe la arquitectura del sistema

VoDKATV, mostrando los componentes software principales que forman parte de dicho sistema, así como las partes del mismo que se usarán como ejemplos de aplicación para las metodologías y técnicas de pruebas desarrolladas en este trabajo. Finalmente, se realizará un resumen de este capítulo en la sección 3.6.

3.2. La televisión interactiva: nuevos servicios multimedia *online*

En los últimos años, la televisión ha estado experimentando una evolución constante. Por un lado, se ha producido un cambio desde la televisión analógica tradicional a la televisión digital. Por otro lado, los televisores de tubo tradicionales han sido substituidos por los de plasma, los cuales posteriormente dieron paso a los LCD, y éstos, a su vez, han ido evolucionando hacia la tecnología LED.

De todas formas, el cambio más importante sufrido en la última década va dirigido hacia la *televisión interactiva*. De esta forma, se pretende cambiar el concepto de televisión tradicional, en la que el usuario final ve la televisión como un aparato que puede usar para ver contenidos determinados de antemano. En este caso, el usuario únicamente puede seleccionar uno de entre varios canales con una programación establecida, pero sin capacidad alguna de decisión sobre el contenido de dichos canales o su programación temporal. La idea principal es cambiar esta comunicación unidireccional por una interacción bidireccional entre la televisión y el usuario, transformando, así, la televisión en un sistema interactivo en el que el usuario pueda decidir qué quiere ver, cuándo, cómo, e incluso dónde quiere verlo.

Y es que el uso de televisores tradicionales está siendo substituido por otros aparatos para "ver la televisión", como son los ordenadores, las tabletas o los móviles. Uno de los puntos clave en este cambio es que los usuarios pueden disfrutar de un entorno completamente personalizado si usan sus propios ordenadores (portátiles o de sobremesa), sus propias tabletas o sus propios teléfonos móviles, puesto que estos tipos de dispositivos suelen ofrecer más posibilidades a la hora de "ver la televisión". Con respecto a las televisiones, éstas están evolucionando hacia las *televisiones inteligentes* (o *Smart TVs*), que ofrecen al usuario muchas más posibilidades que simplemente ver canales de televisión, como, por ejemplo, navegar por Internet, acceder a aplicaciones, redes sociales, pausar la reproducción para continuarla posteriormente, conectar dispositivos multimedia externos, etc.

En cualquier caso, poder visualizar los contenidos de televisión de una forma más personal, adaptando el visionado a los horarios particulares, es a lo que está tendiendo el mundo de la televisión actualmente. Esto es lo que se conoce como *televisión a la carta*. De hecho, la consecuencia de este cambio es una caída en el consumo tradicional de la televisión y un crecimiento cada vez más pronunciado en el consumo de la televisión a la carta.

Un ejemplo donde se puede apreciar este cambio de tendencia es el reproductor *iPlayer* [12] de la BBC [11] (ver figura 3.1). *iPlayer* es una plataforma de televi-

sión a través de Internet que permite visualizar todos los contenidos ofrecidos por la BBC, que incluyen todos aquellos contenidos que se emiten en los canales de la BBC de manera "tradicional" y, en circunstancias especiales, primicias que se estrenan a través de Internet o cortometrajes producidos exclusivamente para Internet. En concreto, se ha pasado de 191 millones de peticiones en Enero de 2012 [104] a 315 millones en Enero de 2014 [103]. Además, aunque inicialmente esta plataforma se usaba sobre todo desde ordenadores (un 66 % de peticiones), actualmente ha aumentado considerablemente el número de dispositivos móviles y tabletas que usan la plataforma, hasta tal punto que en Abril de 2014, por primera vez, el uso de las tabletas (30 %) ha superado al uso de los ordenadores (28 %) para visualizar programas de televisión.

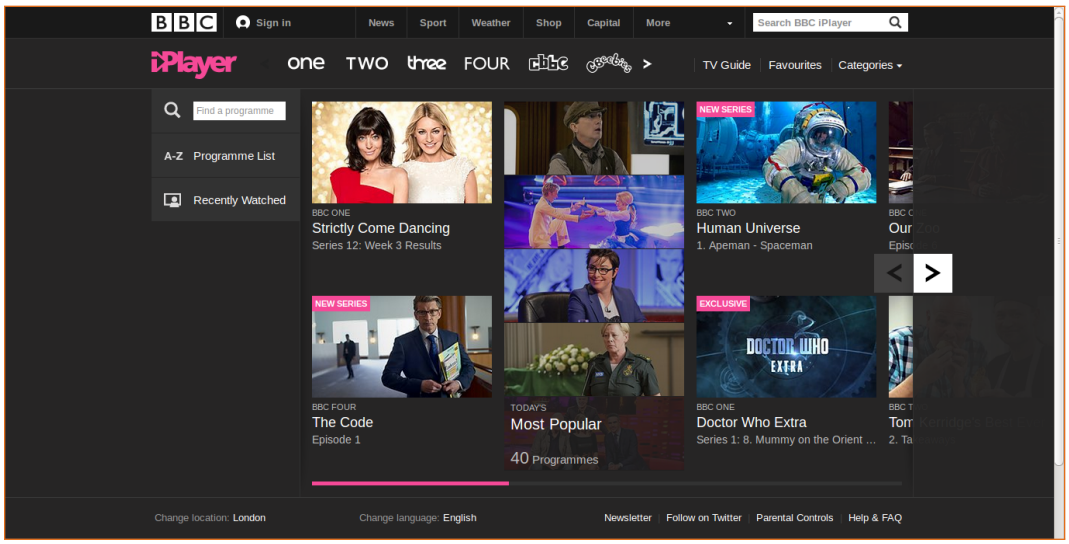


FIGURA 3.1: *iPlayer de la BBC [12]*

Este hecho supone una verdadera oportunidad para las aplicaciones interactivas para la televisión, las cuales, como VoDKATV, buscan ofrecer una experiencia multimedia personalizada al usuario final, ofreciendo aplicaciones de valor añadido a la visualización de contenidos multimedia en múltiples dispositivos.

3.3. La televisión digital: los entornos IPTV y OTT

El término *televisión digital* se refiere al sistema de telecomunicaciones para emitir y recibir imágenes en movimiento y sonidos a través de señales digitales. Este sistema se ha desarrollado intensamente en los últimos años hasta el punto de substituir casi totalmente a la señal analógica tradicional. Así, estándares como PAL, NTSC o SECAM, usados para el vídeo analógico, han dado paso a otro tipo de estándares que rigen la televisión digital, como son ATSC, DVB o ISDB.

Centrándose en el marco europeo, el organismo DVB [19] ha creado un conjunto de estándares aceptados para la televisión digital. Estos estándares, mantenidos por el propio organismo DVB, se utilizan mayoritariamente en Europa, aunque también se usan en otros continentes. Así, este organismo, impulsado por un consorcio de aproximadamente 270 empresas de radiodifusión y distribuidores de equipamiento europeos, como Nokia, Siemens o la BBC, se encarga de proponer y crear los procedimientos de estandarización para la televisión digital. En concreto, el organismo DVB ha producido más de 40 estándares [20] desde que fue fundado en 1993. Entre los estándares proporcionados por este organismo, en función de las características del sistema de radiodifusión, se encuentran:

- DVB-C y DVB-C2 que describen las transmisiones de señales de televisión digital mediante redes de cable,
- DVB-H para televisión terrestre para dispositivos portátiles,
- DVB-IP que fue creado para la transmisión de servicios multimedia utilizando la red IP,
- DVB-S y DVB-S2 para señales de televisión digital mediante redes de distribución por satélite,
- DVB-SH para televisión por satélite para dispositivos portátiles,
- DVB-T y DVB-T2 que definen la televisión digital terrestre.

Con respecto al estándar DVB-IP, las redes IP son el medio de transmisión de mayor interés para el caso de estudio utilizado en este trabajo, aunque no está limitado a únicamente a este tipo. Relacionado con el uso de redes de comunicación IP y la televisión digital, se encuentran los dos entornos más importantes actuales: los sistemas IPTV y los sistemas OTT, los cuales serán explicados a continuación.

3.3.1. Los sistemas IPTV

Los sistemas de televisión que utilizan la distribución de contenidos a través del protocolo IP se conocen como sistemas IPTV. Los tipos de contenidos que se suele distribuir en este tipo de sistemas son:

- Televisión en directo, que puede contener contenidos de pago para los que es necesario realizar un abono para poder visualizarlos.
- Televisión en diferido, es decir, visualizar un contenido del pasado (*catch-up TV*), ver el contenido actual desde el principio (*start-over TV*), o pausar la reproducción del canal para continuarla más adelante.
- Vídeo bajo demanda, por ejemplo, seleccionar una película de un catálogo de vídeo.

Además de visualizar contenidos, en los sistemas IPTV, los usuarios pueden interactuar con la televisión a través de aplicaciones, como pueden ser juegos, servicios de información, compras, mensajería, acceso a Internet, etc.

Por otra parte, es importante señalar que IPTV no significa televisión a través de Internet. Si bien es cierto que en ambos casos se utiliza el protocolo IP, en los sistemas IPTV se utilizan redes IP con calidad de servicio, en contraposición a Internet, que es una red sin calidad de servicio. De esta forma, los sistemas IPTV están pensados para ser usados en redes que aseguren una determinada disponibilidad de recursos disponibles.

La figura 3.2 muestra la arquitectura básica de un sistema IPTV. Por un lado, es posible observar los distintos tipos de redes que forman parte de un despliegue IPTV, comenzando por la red del hogar, desde la que se usa una red de acceso para conectarse a la red de distribución, a donde a su vez llegan los contenidos procedentes de diversas fuentes a través de la red de contribución.

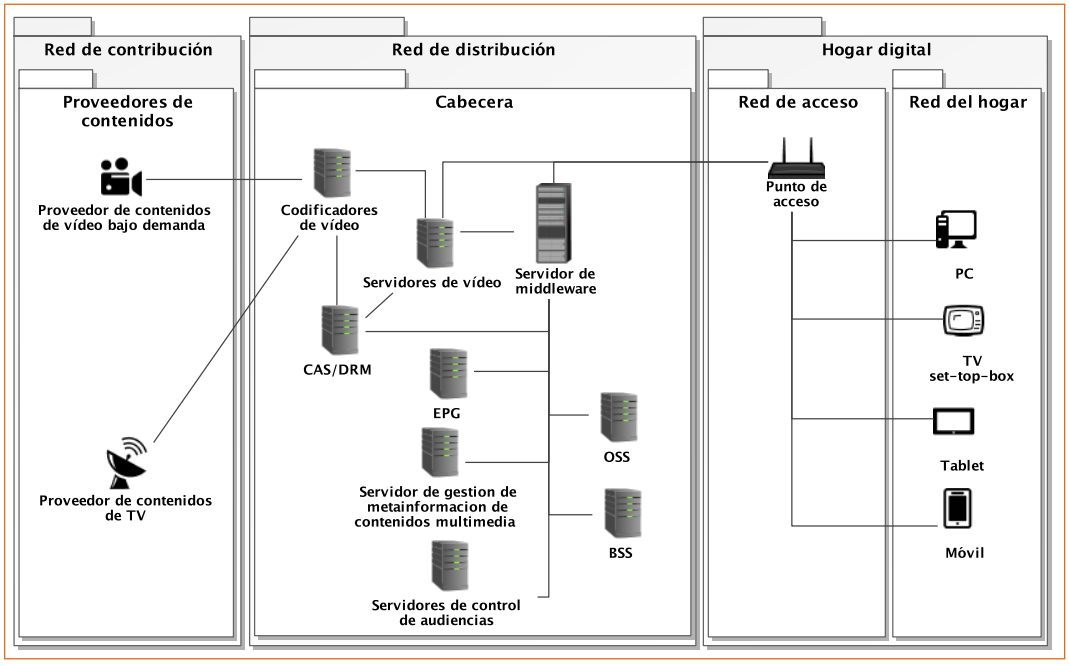


FIGURA 3.2: *Arquitectura típica de los sistemas IPTV*

Uno de los componentes principales de la arquitectura típica de los sistemas IPTV es la cabecera de televisión, la cual obtiene los canales multimedia de diferentes fuentes (contenido local analógico, vídeo analógico o digital a través de satélite, etc.) y lo trasmite a través de una red IP con algún tipo de codificación estándar como son MPEG-2 o MPEG-4. Por otro lado, y al mismo nivel que la cabecera de televisión, en los despliegues IPTV suele haber otros componentes, como, por ejemplo:

- el sistema CAS/DRM, que proporciona protección y derechos a los contenidos multimedia para evitar su difusión indebida,
- servidores de vídeo bajo demanda,
- proveedores de EPG de canales de televisión,
- servicios de facturación BSS,
- sistemas de provisión de usuarios y servicios OSS,
- sistema de control de audiencias,
- etc.

Con respecto a los dispositivos requeridos en el hogar para poder acceder a los servicios ofrecidos por el sistema IPTV se encuentran los ordenadores, las tabletas, los móviles o las televisiones. Este último suele el caso más común en este tipo de sistemas, donde se necesita un dispositivo conocido como *set-top-box* para su funcionamiento. El *set-top-box*, terminal de cliente o descodificador de cliente, es un dispositivo que permite a un aparato de televisión recibir y descodificar las difusiones de televisión digital. Adicionalmente, estos dispositivos suelen disponer de interfaces que red que permiten, por ejemplo, conectarse a Internet, lo que proporciona automáticamente interactividad a la televisión (canal de retorno), o recibir vídeo o audio que puede ser reproducido por el propio dispositivo. Este último tipo de dispositivos se conocen como *set-top-boxes* IP. Cabe destacar también que los *set-top-boxes* pueden ser usados en ámbitos diferentes a los entornos IPTV.

En cualquier caso, en general, los *set-top-boxes* actuales suelen ser dispositivos con pocos recursos, con un sistema operativo base, como puede ser Linux o Windows CE, y donde es posible ejecutar diferentes tipos de aplicaciones, como pueden ser una guía electrónica de programas, un videoclub o juegos, entre otras. La forma en la que se programan estas aplicaciones interactivas depende de cada *set-top-box*. No obstante, en la arquitectura de los *set-top-boxes* suele existir una capa *middleware*, situada entre el sistema operativo y la capa de aplicaciones, que ofrece una API de desarrollo de aplicaciones de alto nivel. De esta forma, los programadores de aplicaciones no necesitan tener en cuenta detalles de bajo nivel del *set-top-box*, como pueden ser el propio sistema operativo o los controladores, puesto que el *middleware* los abstrae. Así, se facilita la tarea de implementación de aplicaciones para el *set-top-box*.

Algunos ejemplos de plataformas *middleware* son: MHP [177, 179, 298, 299], Java TV [131, 338], OpenTV [45], MediaHighway [15], etc. Cada una de estas plataformas ofrece APIs para el desarrollo de aplicaciones, normalmente en forma de SDK, es decir, un conjunto de herramientas que permiten implementar aplicaciones para el *set-top-box*. Así, por ejemplo, MHP y Java TV ofrecen una API Java [269], en OpenTV se programan las aplicaciones usando el lenguaje de programación C, y

en MediaHighway es posible usar diferentes lenguajes como PanTalk, Java o HTML/JavaScript. Otro ejemplo de *middleware* para *set-top-boxes* (en este caso, *set-top-boxes* IP) es HED [310], desarrollado por Interoud Innovation y LambdaStream usando mayormente el lenguaje de programación Erlang, que ofrece una API JavaScript para el desarrollo de aplicaciones interactivas, las cuales se ejecutan en un navegador web, en concreto, WebKit [67].

Además, en los despliegues IPTV es necesaria una pieza que conecte los sistemas de la cabecera de televisión con los usuarios, y viceversa. En este contexto, este componente también se conoce como *servidor de middleware*, y debe asegurar la completa interoperabilidad de los servicios ofrecidos por el sistema IPTV, siendo capaz de comunicarse directamente con cada componente involucrado en el despliegue. Por ejemplo, cuando un usuario desea visualizar un contenido del catálogo de vídeo, se realizará una solicitud al *servidor de middleware*, el cual autorizará la operación en función del usuario concreto, comprobando en el sistema BSS si éste tiene crédito disponible, y si esta solicitud es aceptada se comunicará con el servidor CAS/DRM para indicarle que el usuario tiene derecho a visualizar ese contenido durante un período de tiempo, con el servidor de vídeo para indicar que comience la transmisión de la película y, finalmente, con el sistema BSS para completar la compra.

Así, el *servidor de middleware* interactúa con los diferentes subsistemas para permitir la distribución de los servicios de televisión interactivos. Para ello, además de almacenar información de configuración global del sistema, debe identificar a los usuarios o suscriptores (bien almacenándolos por él mismo o bien consultando un servicio externo de OSS), y ofreciendo únicamente los servicios que éstos tienen contratados.

Aunque existen diversos estándares que pueden condicionar la arquitectura de componentes usada en el *servidor de middleware*, éstos no están lo suficientemente extendidos, y las soluciones empleadas son mayoritariamente propietarias. Normalmente, se usa un protocolo basado en HTTP (o HTTPS), mediante el cual se intercambian datos en formato HTML, JSON o XML para comunicar el *servidor de middleware* con el cliente, es decir, el dispositivo usado por el usuario como, por ejemplo, un *set-top-box*.

3.3.2. Los sistemas OTT

Como ya se ha mencionado, los sistemas IPTV se usan en redes en las que el sistema completo está bajo el control de un operador o proveedor de servicios, el cual, normalmente, cobra una suscripción a sus usuarios, y además garantiza la calidad del servicio gracias al uso de una red privada propia. Sin embargo, este hecho provoca que los costes de despliegue y mantenimiento sean altos, además de problemas de escalabilidad. Es por ello que en muchos entornos se está migrando

esta arquitectura hacia otra centrada en el uso de Internet como red de distribución. Este paradigma se conoce como OTT.

Así, el término OTT se refiere a la distribución de vídeo, audio y otros contenidos multimedia a través de Internet sin un operador concreto involucrado en el control o distribución de los contenidos. Casos como Netflix [46], YouTube [78] o el reproductor *iPlayer* [12] de la BBC son ejemplos de plataformas OTT que permiten a los usuarios disfrutar de contenidos multimedia en sus dispositivos (ordenadores, tabletas, móviles, *set-top-boxes*, videoconsolas, etc.) sin necesidad de un operador que ofrezca estos servicios multimedia. Además, mientras que el acceso a los sistemas IPTV está limitado por el operador, los sistemas de OTT pueden ser accedidos desde cualquier ubicación en Internet, obviamente teniendo en cuenta las limitaciones de uso que puede tener el sistema por localización geográfica, o el registro y autenticación de usuarios.

Teniendo en cuenta este cambio de paradigma de IPTV a OTT, es lógico que los protocolos de comunicación usados en ambos casos sean ligeramente diferentes. Así, en los sistemas IPTV se usa el protocolo de multidifusión (*multicast*) para emitir los canales de televisión o radio que serán recibidos por los descodificadores de cliente. De esta forma, únicamente se genera un único flujo de vídeo independientemente del número de clientes que estén sintonizando dicho canal. Actualmente, este protocolo no está soportado en Internet y, en su lugar, en este tipo de redes se usan conexiones desde un único emisor a un único receptor (*unicast*) tanto en la emisión de canales como en vídeo bajo demanda. Este hecho provoca que se utilice normalmente el protocolo HTTP para la transmisión de datos, usando descarga progresiva y empleando protocolos de transmisión adaptativos como HLS, *Smooth Streaming* o DASH.

En algunos despliegues se usa un enfoque mixto, en el que se usan las técnicas de OTT para contenidos bajo demanda y canales menos populares, mientras que se usa multidifusión IPTV para los canales de televisión más importantes. Esto es así porque IPTV continúa siendo la mejor forma de garantizar la disponibilidad del servicio y la seguridad de los contenidos bajo redes IP. Por otro lado, Internet permite distribuir contenidos multimedia ilimitados, como canales de televisión, a un número ilimitado de usuarios, aunque sin garantizar la calidad de servicio. Obviamente, el ancho de banda se debe tener en cuenta por motivos de escalabilidad.

La tabla 3.1 muestra un resumen comparativo entre ambas aproximaciones.

3.4. El proyecto VoDKATV: una solución para entornos IPTV y OTT

El proyecto VoDKATV surge en el año 2006 con el objetivo de crear un sistema IPTV que pueda ser usado en hoteles para ofrecer servicios multimedia a sus clientes a través de las televisiones de sus habitaciones. Así, VoDKATV nació como un

Criterio	IPTV	OTT
Tipo de red	Redes dedicadas y gestionadas	Internet
Calidad de servicio	Control de calidad en la entrega	No garantizada
Protocolos	RTSP, HTTP, UDP	HTTP: HLS, *Smooth Streaming*, DASH
Difusión de contenidos	*Unicast* y *multicast*	*Unicast*
Beneficios	Calidad de servicio	Bajo coste. Flexibilidad de consumo de contenidos a través de diferentes dispositivos
Inconvenientes	Coste alto	Baja calidad de servicio
Ejemplos	Verizon FiOS [73], AT&T Uverse [10]	Netflix [46], YouTube [78], *iPlayer* [12] BBC

TABLA 3.1: *Resumen comparativo entre IPTV y OTT*

sistema cliente-servidor, donde los clientes eran únicamente *set-top-boxes* IP. En concreto, inicialmente se usaron *set-top-boxes* de la marca Amino [5], los cuales llevaban incorporado un navegador que se usaba para mostrar la interfaz de usuario.

Con el tiempo, este proyecto ha ido creciendo, pues ha sido adoptado, inicialmente por la empresa LambdaStream y, posteriormente, por Interoud Innovation, como uno de sus productos principales. Equipos de trabajo de entre 3 y 5 personas han estado trabajando en el desarrollo y evolución de este proyecto desde el año 2007 hasta la actualidad. Actualmente, el código de VoDKATV consta de más de 250K LOC Java, repartidas en más de 3K clases Java, aparte del código HTML, JavaScript y CSS usado en las aplicaciones cliente.

De esta forma, VoDKATV se ha convertido en un *servidor de middleware* que puede ser usado tanto en entornos IPTV como en entornos OTT, con soporte para más tipos de *set-top-boxes* que inicialmente, incluyendo el propio *middleware* HED. El sistema VoDKATV integra información de diferentes componentes externos, como son el servidor de BSS, OSS, servidor de metainformación de contenidos bajo demanda, o el servidor de EPG, entre otros; y ofrece una API HTTP (o HTTPS) para que aplicaciones externas, ejecutadas en dispositivos de cliente como son ordenadores, tabletas, móviles o *set-top-boxes* IP, puedan usar la información que el propio servidor VoDKATV proporciona. Cuando el sistema VoDKATV se utiliza en entornos OTT, éste se encuentra ubicado en Internet, y el usuario se descarga o accede a una aplicación a través de un dispositivo propio, que realizará conexiones al servidor de VoDKATV cuando necesite datos almacenados remotamente. Por el

contrario, en entornos IPTV, tanto el servidor VoDKATV como los clientes, están ubicados en la misma red.

El sistema VoDKATV proporciona una serie de funcionalidades que permiten ofrecer a los usuarios una experiencia multimedia en múltiples dispositivos, entre las que destacan:

- Acceso a una lista de canales de televisión y de radio "en directo" (libres y de pago), los cuales pueden llegar al sistema a través diferentes fuentes como son satélite, cable, Internet o incluso usando ficheros multimedia almacenados; con acceso a la lista de programas, múltiples audios, subtítulos y teletexto.
- Acceso a funcionalidades de reproducción de televisión en diferido: ver un programa del pasado (*catch-up TV*), comenzar el programa actual desde el principio (*start-over TV*), y pausar la reproducción del canal actual para continuarla más adelante.
- Grabación local de canales de televisión (PVR).
- Acceso a un catálogo de vídeo con contenidos de pago.
- Reproducción de contenidos locales (vídeos, audios e imágenes) y compartidos en redes locales (por ejemplo, usando Samba o NFS).
- Acceso a una tienda: compra de paquetes, productos y aplicaciones adicionales.
- Navegación por Internet.
- Juegos.
- Consulta de facturas.
- Configuración de preferencias personales: idioma, control parental, configuración de red, configuración de canales favoritos, etc.

En la actualidad, el sistema VoDKATV se encuentra instalado en diferentes entornos. Así, está siendo usado en hoteles en Oriente Medio, como, por ejemplo, Diva Hotel [66], Salmiya Millennium Hotel [44] (ver figura 3.3) o Ramada Hotel [60], entre otros. También se han realizado despliegues sobre entornos de telecomunicaciones, como por ejemplo, TalkEasy [63], un proveedor suizo de servicios IPTV; EWETEL [28] (ver figura 3.4), compañía eléctrica alemana que ofrece servicios OTT a sus usuarios; o Netcologne [64], una compañía alemana que ofrece servicios de telefonía, televisión e Internet. Por otra parte, VoDKATV también se está usando en otros despliegues relacionados puramente con televisión a través de Internet, entre ellos, el portal de televisión por Internet de la Universidad de A Coruña [71], UDCTV [72] (ver figura 3.5). Por último, destacar que VoDKATV también actúa como sistema de *back-end* (puesto que la interfaz de usuario es proporcionada por

otros componentes externos al propio sistema VoDKATV) en otros despliegues como es el sistema OTT proporcionado por la compañía gallega R [59].

FIGURA 3.3: *Interfaz de usuario proporcionada por VoDKATV para el cliente Millennium Hotel visualizada en un set-top-box HED (pantalla de inicio)*

3.5. Arquitectura del sistema VoDKATV

VoDKATV ha sido diseñado como un sistema compuesto por una serie de componentes software independientes. Cada uno de estos componentes proporciona un conjunto de interfaces, las cuales definen las operaciones públicas que otros componentes deben usar para acceder a la información que estos componentes manejan. El diagrama UML que se muestra en la figura 3.6 muestra los componentes principales de la arquitectura del sistema VoDKATV, así como las interacciones entre ellos. VoDKATV es un sistema cliente-servidor en el que la parte cliente la componen los diferentes dispositivos que acceden al sistema (ordenadores, tabletas, móviles, *set-top-boxes*, etc.), los cuales acceden a la información almacenada en la parte servidor.

En concreto, estos dispositivos se comunican principalmente con el componente VoDKATV-core, que es el componente principal de la solución VoDKATV. Este componente es una aplicación J2EE que se instala en un contenedor de aplicaciones web (usualmente Apache Tomcat [9]), y usa una base de datos relacional (PostgreSQL [55]) que almacena información necesaria para el funcionamiento del sistema. Este componente se corresponde con el *servidor de middleware* que se muestra

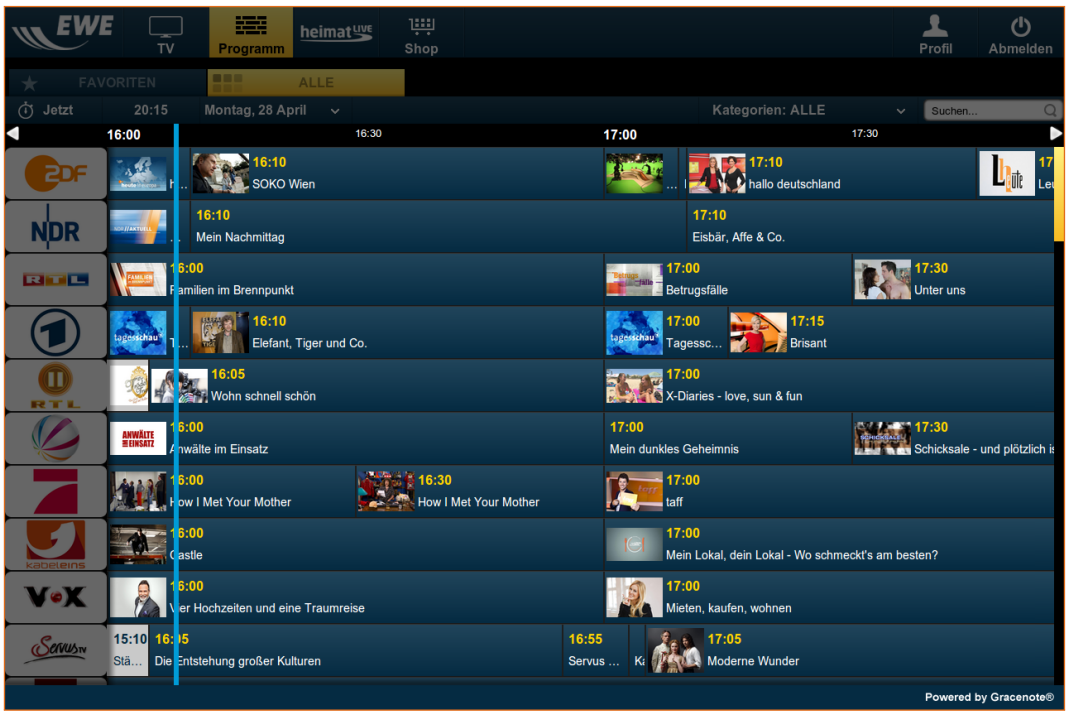


FIGURA 3.4: *Interfaz de usuario proporcionada por VoDKATV para el cliente EWETEL visualizada en un ordenador usando el navegador Chrome (guía electrónica de programas)*

en la figura 3.2 (la cual muestra de una forma genérica los componentes involucrados en un despliegue típico de IPTV), con la peculiaridad de que el componente VoDKATV-core contiene implementaciones propias de los módulos OSS y BSS que se usarán en aquellos despliegues en los que estos sistemas no sean proporcionados por un tercero.

Como se observa en la figura 3.6, el componente VoDKATV-core también se integra con otros componentes, los cuales pueden ser agrupados como componentes de *back-end*:

- Servidores de vídeo, que actualmente pueden ser servidores de vídeo Wowza [74], o servidores de vídeo VoDKA [200]. Además, cuando se usa VoDKA como servidor de vídeo, es posible usar otro componente, llamado LiveScheduler, para gestionar directos y grabaciones.
- Un servidor de gestión de metainformación de contenidos multimedia, desarrollado por Interoud Innovation, y llamado VoDKA Asset Manager.
- Un servidor de EPG, implementado íntegramente por Interoud Innovation, y que permite obtener la guía de programación de canales a través de diversas fuentes, como, por ejemplo, XMLTV [76] u otras fuentes propietarias propor-

FIGURA 3.5: *Interfaz de usuario proporcionada por VoDKATV para el cliente UDCTV visualizada en un ordenador usando el navegador Firefox (listado de contenidos)*

cionadas por terceros.

- Un sistema de CAS/DRM, en concreto, se han realizado integraciones con el sistema de CAS/DRM proporcionado por la compañía Irdeto [36] y con el sistema Windows Media DRM.
- Un sistema de control de audiencias, que puede ser, en función de la instalación, Google Analytics [30], Piwik [54] o un componente propio implementado por Interoud Innovation para almacenar y recuperar información de uso del sistema (canales más vistos, usuarios conectados, tipos de dispositivos usados, etc.).

Por otro lado, en el sistema VoDKATV también están presentes una serie de aplicaciones de administración que permiten a los administradores configurar el sistema. Esta configuración del sistema consiste en definir, por ejemplo, los canales que tendrán disponibles los usuarios, la configuración del catálogo de vídeo, los precios de los diferentes elementos adquiribles como son los paquetes, productos o aplicaciones, y, en definitiva, todas aquellas acciones necesarias para que los usuarios puedan disfrutar de los servicios multimedia.

Las partes del sistema VoDKATV que serán usadas como ejemplo para ilustrar, de una manera práctica, el uso de las distintas metodologías y técnicas de pruebas desarrolladas en este trabajo son las siguientes:

- En el capítulo 5 se usará una librería de plantillas correspondiente a código fuente Erlang del *set-top-box* HED.

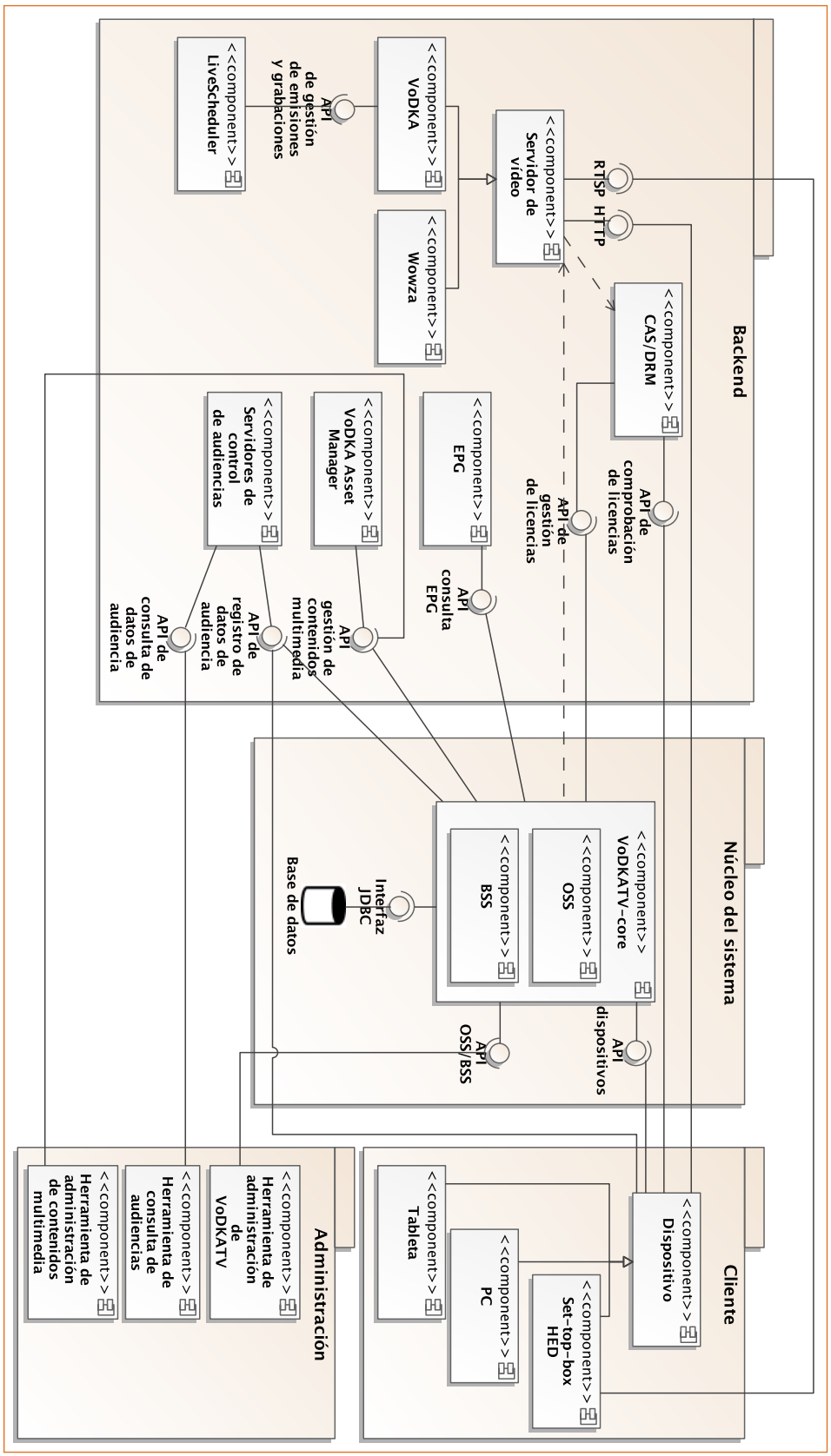


FIGURA 3.6: *Arquitectura del sistema VoDKATV*

- En el capítulo 6 se probará la API de integración proporcionada por el componente VoDKA Asset Manager para la gestión de contenidos multimedia.
- En el capítulo 7 se utilizará la API de integración OSS/BSS proporcionada por el componente VoDKATV-core usada por las aplicaciones de administración para ilustrar cómo probar un servicio web que maneja datos en XML.
- En el capítulo 8 se probará la integración entre el componente LiveScheduler y el componente VoDKA para gestionar emisiones y grabaciones de canales multimedia.
- En el capítulo 9 se evaluará el rendimiento del componente VoDKATV-core con respecto al número de usuarios soportados por la plataforma, para lo que se usará la misma API de integración que usan los dispositivos para comunicarse con dicho componente.

Como se observa, utilizando como caso de estudio un único sistema software real, es decir, el sistema VoDKATV, es posible ilustrar cómo aplicar las metodologías y técnicas de pruebas descritas en cada uno de los capítulos de este trabajo, puesto que cada una de ellas servirá para probar diferentes aspectos de un sistema software.

3.6. Resumen

Este capítulo describe el sistema VoDKATV, que se usará como caso de estudio para ilustrar la aplicación de todas las metodologías y técnicas desarrolladas en este trabajo. Como se observa, el sistema VoDKATV es un sistema complejo, compuesto por varios componentes que se integran entre sí para lograr una serie de funcionalidades, relacionadas con el acceso a contenidos multimedia y aplicaciones interactivas a través de múltiples dispositivos.

El hecho de que VoDKATV sea ser un sistema real usado en despliegues reales convierte a este ejemplo en un caso de estudio de gran valor para ilustrar las metodologías y técnicas desarrolladas en este trabajo. Además, su gran tamaño, junto con la propia naturaleza del sistema por la arquitectura que tiene en respuesta al entorno de negocio para el que se construye, hace que sea sencillo encontrar todos y cada uno de los casos para los cuales dichas metodologías y técnicas de pruebas han sido creadas. De hecho, como se explicará a lo largo de este trabajo, muchas de las partes del sistema VoDKATV han podido ser probadas con las técnicas propuestas.

Evidentemente, estas metodologías y técnicas no han sido diseñadas exclusivamente para probar el sistema VoDKATV, sino que pueden ser usadas en otros sistemas software. No obstante, el uso de un sistema complejo para ilustrar la aplicación práctica de estas metodologías y técnicas, como es VoDKATV, permite, además de entender mejor cómo se usan, hacerse una idea del tipo de sistemas software en los que dichas metodologías y técnicas pueden ser utilizadas.

4

Análisis y diseño

4.1. Introducción

Las fases de análisis y diseño son fundamentales en el desarrollo de software. De hecho, como se comenta en el capítulo 2, las primeras etapas de cualquier ciclo de vida se dedican a realizar este tipo de tareas. Estas fases permiten definir, a partir de unos requisitos iniciales, qué hay que hacer en un proyecto software y cómo se debe estructurar el mismo para llevar a cabo dicho proyecto.

Al igual que en las tareas de implementación, en las fases de análisis y diseño también es posible introducir defectos que provocarán que el sistema a implementar no funcione como el cliente espera. De hecho, existen estudios que afirman que aproximadamente dos terceras partes de los defectos encontrados se deben a errores en las fases iniciales de análisis y diseño [336]. Además, como se menciona en el capítulo 1, cuánto más se tarde en detectar estos defectos, más costoso será solucionarlos, por lo que los defectos introducidos en las fases de análisis y diseño pueden llegar a ser los más costosos de solucionar si no se detectan a tiempo. Es por ello que es importante realizar revisiones e inspecciones que permitan comprobar que las especificaciones resultantes del análisis y el diseño se corresponden con los requisitos iniciales establecidos por el cliente [292].

Este capítulo describe en qué consisten las fases de análisis y diseño de sistemas software a partir de unos requisitos iniciales, mostrando cómo encajan las tareas de pruebas dentro de dichas fases. Para ello, se explican los conceptos básicos de cada una de estas fases en la sección 4.2, enfatizando los aspectos a tener en cuenta a la hora de llevar a cabo las tareas de análisis y diseño de sistemas informáticos para

que el software resultante tenga una alta calidad y pueda ser probado de manera eficiente y eficaz. Posteriormente, se realiza una breve introducción a los lenguajes de modelado en la sección 4.3. En concreto, se introduce el lenguaje de modelado UML, el cual es una herramienta estándar para capturar y representar la información resultante del análisis y diseño de sistemas software. Por último, en la sección 4.4, se realiza un resumen de los contenidos de este capítulo.

4.2. De los requisitos al análisis y diseño

Esta sección describe las actividades que forman parte del proceso de desarrollo de software desde la obtención de los requisitos hasta la definición de un diseño que especifique, en detalle, cómo debe estructurarse el software que se va a construir.

Describir este proceso tiene dos objetivos principales. Por un lado, se muestra cómo las actividades de pruebas están presentes desde el inicio del ciclo de vida de un producto software, incluyendo actividades de revisión e inspección de, al menos, los requisitos, análisis y diseño del sistema. Por otro lado, introducir estas actividades permitirá describir, como se comentará más adelante, aquellos aspectos que se deben tener en cuenta para que el objetivo de realizar pruebas automáticas de una manera cómoda sea realizable.

4.2.1. Los requisitos del software

Los requisitos del software se corresponden con las especificaciones, acordadas normalmente en las primeras etapas del desarrollo de un sistema software, que indican lo que debe ser implementado, describiendo cómo se debe comportar el sistema [330]. Por ejemplo, una funcionalidad de usuario, una propiedad general del sistema o una restricción del sistema son requisitos del software.

Existen numerosos tipos de requisitos [328], aunque de manera general éstos pueden ser clasificados en requisitos funcionales y requisitos no funcionales. Los requisitos funcionales describen lo que el sistema debe hacer, es decir, comportamientos observables del sistema bajo determinadas condiciones. Por ejemplo, un requisito como “los usuarios deben ser capaces de consultar su factura mensual a través de la televisión” es un requisito funcional. Por otro lado, los requisitos no funcionales especifican características o propiedades que el sistema debe tener, o bien restricciones que se deben cumplir, y en vez de especificar qué hace un sistema, suelen especificar cómo lo hace. Por ejemplo, la disponibilidad de un sistema, la usabilidad, escalabilidad, seguridad, rendimiento, o incluso restricciones de diseño, implementación o entornos de instalación forman parte de los requisitos no funcionales de un sistema software.

La *ingeniería de requisitos* define el proceso de formular, documentar y mantener los requisitos del software. A su vez, este campo puede ser dividido en dos

partes [366] (ver figura 4.1): la *gestión de requisitos*, que incluye todas las actividades que mantienen la integridad de los requisitos a lo largo de todo el proyecto software; y el *desarrollo de requisitos*, que abarca todas las actividades que permiten obtener, evaluar, documentar y confirmar los requisitos de un sistema software.

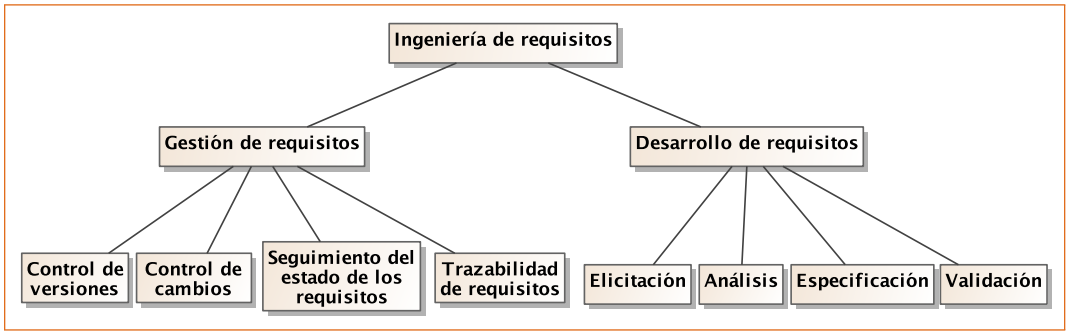


FIGURA 4.1: *Subdisciplinas de la ingeniería de requisitos*

Dentro la gestión de requisitos existen una serie de actividades a realizar, que se agrupan en las siguientes categorías principales:

- **Control de versiones**, es decir, la definición de un sistema de versionado y la asignación de versiones a cada uno de los requisitos.
- **Control de cambios**, que incluye todas aquellas actividades desde que se propone un cambio en los requisitos, se analiza el impacto de los mismos y se toman decisiones acerca de los cambios, hasta que se actualizan los requisitos a cambiar.
- **Seguimiento del estado de los requisitos**, esto es, la definición de los posibles estados que se pueden asociar a los requisitos, así como el registro del estado de cada uno de ellos.
- **Trazabilidad de requisitos**, que incluye el establecimiento de relaciones y dependencias entre los requisitos, y de los requisitos con otros elementos del sistema.

Aunque una mala gestión de requisitos podría causar estragos en el software resultante, es en las actividades del área de desarrollo de requisitos donde es posible realizar tareas de pruebas para prevenir problemas que puedan causar que el software a implementar sea defectuoso. Las tareas que forman parte del área del desarrollo de requisitos son las siguientes:

- **La elicitación de requisitos**, que engloba todas las actividades relacionadas con la captura de requisitos. Existen diferentes técnicas de elicitación de requisitos [196], como son las entrevistas, técnicas de prototipado, talleres de trabajo, observación, etc.

- **El análisis de requisitos**, cuyo objetivo principal es entender cada uno de los requisitos existentes, definiendo los límites de cada uno de ellos. Además, también se detectan y resuelven conflictos entre los diferentes requisitos. Para ello, los requisitos se clasifican en diferentes tipos, y los requisitos de alto nivel se detallan y se descomponen en requisitos de más bajo nivel si es necesario.
- **La especificación de requisitos**, que consiste en representar los requisitos existentes de tal forma que puedan ser revisados por todas las partes implicadas en su desarrollo. Así, como resultado de esta tarea, se obtendrá documentación asociada al desarrollo de requisitos, como es el documento de especificación de requisitos del software.
- **La validación de requisitos**, cuyo objetivo principal es asegurarse de que los requisitos obtenidos se corresponden con aquellos que permitirán obtener como resultado el sistema software deseado. De esta forma, existen diferentes frentes que deben ser comprobados. Por un lado, es necesario comprobar que la documentación de requisitos es entendible, consistente y completa como para que el equipo de desarrollo construya el software objetivo. Por otro lado, se debe comprobar que la documentación de requisitos describe correctamente los requisitos especificados por el cliente. Para realizar estas comprobaciones, las técnicas más usadas [156, 328, 329, 344] son las revisiones de los requisitos, el prototipado, o la elaboración de pruebas de aceptación, las cuales permiten comprobar que el producto software resultante cumple con los requisitos iniciales establecidos.

Es muy importante que los procesos relacionados con los requisitos del software se realicen con toda la rigurosidad necesaria. De hecho, los propios requisitos están relacionados con muchas de las causas más habituales por lo que los proyectos fracasan [173, 221, 352], como es la obtención de una especificación de requisitos incompleta o imprecisa.

En resumen, los requisitos deben ser claros, específicos y no deben ser ambiguos. Para ello, deben ser medibles en términos de valores específicos, y debe ser posible comprobar que éstos se cumplen con algunos criterios de evaluación. En conjunto, la especificación de requisitos debe ser completa y no establecer contradicciones entre los requisitos que la componen. Esto es importante puesto que si el software se basa en requisitos incorrectos, incluso aunque el código se escriba correctamente, el resultado final no será satisfactorio.

4.2.2. El análisis del software

A partir de los requisitos identificados, el objetivo de esta fase es obtener una serie de modelos de análisis del software que describan, de manera no ambigua, qué es lo que el sistema debe hacer. Así, detalles técnicos como la arquitectura del

sistema software, que especifica las interfaces con otros elementos del sistema, u otras restricciones que el sistema debe cumplir, como el tipo de aplicación (web o programa *stand-alone*), el sistema operativo en el que podrá usarse la aplicación, las necesidades de hardware, parámetros de concurrencia o eficiencia, entre otros aspectos, se derivan de esta fase a partir del trabajo de los analistas de software.

Cuando se realiza el análisis se debe representar y entender el dominio de información del problema, se deben definir las funciones que debe realizar el software y, por último, se debe representar el comportamiento del mismo [301]. De esta forma, se obtienen una serie de modelos que representan información, función y comportamiento. El principal objetivo de los modelos de análisis es entender mejor la entidad que se va a construir.

Estos modelos de análisis, los cuales son realmente la primera representación técnica del sistema, serán usados por los diseñadores de software, sirviendo como base para la creación del diseño software. De esta forma, el análisis es una fase que se encarga de conectar la fase de requisitos del sistema con la fase de diseño.

Existen diversos métodos para llevar a cabo el modelado de análisis, pero actualmente predominan dos: el análisis estructurado [158] y el análisis orientado a objetos. Con respecto al análisis orientado a objetos, se crearon una serie de métodos como el método de Booch [125], el método de Rumbaugh [314], el método de Jacobson [216], el método de Coad y Yourdon [147] o el método de Wirfs-Brock [368], entre otros. A finales de los años 90, como resultado de combinar y recopilar las mejores características de varios métodos de diseño y análisis orientado o objetos, se creó el *lenguaje de modelado unificado* (UML) [126, 217, 315], el cual se ha convertido en el método más utilizado por la industria tanto en las tareas de análisis como en las de diseño.

4.2.3. El diseño del software

El diseño es el proceso de definir la arquitectura, componentes, interfaces y otras características de un sistema o componente software [1]. Así, la fase de diseño parte del resultado del análisis, tomando como partida los modelos de análisis, para obtener una representación detallada del sistema a construir, esto es, unos modelos de diseño. Estos modelos de diseño describen cuatro áreas del software: la estructura de datos, la arquitectura del sistema, la representación de la interfaz y los detalles a nivel de componente [301].

De esta forma, las actividades de diseño ayudan a construir el software apropiado a partir de los requisitos. No realizar la fase de diseño tiene el riesgo de desarrollar un sistema inestable, difícil de cambiar e incluso en el que es difícil comprobar que el software funciona de la manera deseada.

Existen diferentes paradigmas para llevar a cabo las tareas de diseño del software [372]. Así, desde los enfoques iniciales que se concentran en criterios para el

desarrollo de programas modulares [161] y métodos para refinar las estructuras del software de manera descendente [369], los cuales evolucionaron hacia una filosofía llamada programación estructurada [154], se ha llegado a métodos de diseño más recientes que están orientados a objetos [193].

En cualquier caso, el proceso de diseño consiste en llevar a cabo una serie de pasos que hacen posible que el equipo de diseñadores de software describa todos los aspectos del software que se va a construir. Así, uno de los conceptos de diseño es la definición de la arquitectura del software [324], la cual describe la estructura jerárquica de los componentes que forman parte del sistema software, la manera en la que estos componentes interactúan, y la estructura de datos que van a utilizar en dichas interacciones. La modularidad, es decir, la división del sistema software en componentes o módulos, es uno de los conceptos a tener en cuenta en el diseño, de tal forma que se debe llegar a un compromiso entre el número de módulos y el tamaño de los módulos, con el objetivo de disminuir el coste de desarrollar los módulos sin aumentar el coste de las integraciones entre los mismos [265].

Cuando se definen los módulos o componentes que forman parte de un sistema software, junto con la división entre ellos, se deben tener en cuenta los conceptos de *abstracción* y *ocultación*. Por un lado, la abstracción ayuda a definir las entidades que forman parte del sistema. Por otra parte, el concepto de ocultación significa que la información que maneja un módulo debe ser inaccesible a otros módulos que no necesiten dicha información, y únicamente se usen una serie de interfaces para la comunicación entre módulos, de tal manera que se cumpla la *independencia funcional* entre cada uno de los módulos.

En general, un diseño debe ser modular, es decir, el software debe dividirse lógicamente en elementos que realicen funciones y subfunciones específicas, y debe conducir a interfaces que reduzcan la complejidad de las conexiones entre los módulos y con el entorno externo. Los módulos independientes son más fáciles de mantener y probar porque se limitan los efectos secundarios originados por modificaciones futuras en el diseño y el código, se reduce la propagación de errores, e incluso es posible utilizar módulos reusables. Es por ello que la independencia funcional es la clave para un buen diseño.

Para alcanzar la independencia funcional se deben tener en cuenta dos conceptos [282, 286]: la *cohesión* y el *acoplamiento*. Se dice que un módulo es cohesivo si éste se encarga de realizar una tarea definida dentro de un procedimiento de software con las mínimas dependencias externas necesarias y manejando un nivel de ocultación apropiado. Por otro lado, el acoplamiento es una medida de interdependencia entre módulos dentro de una estructura de software. Un buen diseño del software busca una máxima cohesión dentro de cada módulo y un mínimo acoplamiento entre módulos [373].

Por último, cabe destacar que en cada uno de los pasos del proceso de software

se realiza un refinamiento en el nivel de detalle de los modelos del software. Y así, a medida que avanza la fase de diseño, se detallan más aspectos del sistema a construir, hasta que se obtiene el nivel más alto de detalle cuando se produce el código fuente.

4.2.4. Los patrones de diseño

En la práctica, cuando se realizan las tareas de análisis y diseño de un sistema software, suelen aparecer una serie de situaciones recurrentes que ya han sido tratadas en el análisis y diseño de otros sistemas software. Este hecho ha provocado que se definan patrones [193] que describen situaciones que ocurren una y otra vez en el mundo del software, proponiendo una solución para cada una de ellas. Así, estas soluciones pueden ser aplicadas en situaciones concretas, y evitan tener que reinventar soluciones de análisis y diseño para problemas que ya han sido resueltos anteriormente de forma satisfactoria.

Usar patrones de diseño no necesariamente implica una mejora de la calidad del software [229, 317]. Sin embargo, un uso adecuado de estos patrones, además de acelerar el desarrollo de software debido al uso de soluciones reusables, sí permite conseguir un mejor diseño de un sistema, resultando, por tanto, en software más fácil de implementar, de mantener y evolucionar.

4.2.5. El diseño para facilitar las pruebas

Otro de los aspectos a tener en cuenta a la hora de diseñar un sistema software es la facilidad para realizar las pruebas del mismo en fases posteriores. Las pruebas son una actividad que consume una cantidad de tiempo y recursos considerable en el desarrollo de software [248, 301, 305] y, por tanto, es importante facilitar esta tarea para que se realice de forma eficiente. Para esto, además de mejorar la especificación y documentación del software, usar técnicas y herramientas de pruebas adecuadas o mejorar el proceso de pruebas, se debe tener en cuenta que el diseño del software (y su correspondiente implementación) influye en gran medida en este aspecto.

Así, se define la *capacidad de prueba* (*testability*) [1] de un sistema software como el grado en el que dicho sistema facilita las actividades de pruebas, siendo un indicativo de la cantidad de esfuerzo necesario para probar un sistema. Si la capacidad de prueba es alta, encontrar fallos en el sistema (si existen) por medio de las pruebas es más fácil. Por otro lado, si la capacidad de prueba es baja, se deben incrementar los esfuerzos para probar el sistema software y, por tanto, también el coste. Diseñar un sistema software teniendo en cuenta las actividades de pruebas se denomina *diseño para facilitar las pruebas* (DFT) [219]. Por tanto, a la hora de diseñar un sistema, además de tener en cuenta si va a ser posible su posterior implementación, es importante tener en cuenta si el sistema va a poder ser probado fácilmente.

Un alto acoplamiento entre componentes, componentes complejos que llevan a cabo muchas tareas, falta de abstracción, no respetar el encapsulado de los componentes, dependencias fijas o cíclicas entre componentes y capas o con recursos externos, son algunos de los aspectos a evitar en el diseño de un sistemas software. Estos puntos, además de lograr como resultado un diseño poco mantenible y resistente a cambios, dificultaría las tareas de pruebas del software resultante.

De esta forma, y teniendo en cuenta las pruebas, el diseño del software debe facilitar este tipo de actividades. En general, para que un sistema software pueda ser probado sin aumentar la dificultad de las pruebas se debe garantizar que todos los componentes del sistema pueden ser probados de forma independiente, los casos de prueba pueden ser identificados y los resultados de las pruebas pueden ser observados. Con respecto a las tareas de diseño, respetar la abstracción, no violar el encapsulado, cuidar las dependencias entre componentes evitando ciclos, y diseñar los componentes con responsabilidades únicas y definidas son aspectos clave que se deben seguir para garantizar un bueno diseño del software, así como una alta capacidad de prueba del mismo. También existen recomendaciones de uso de los patrones de diseño para que la capacidad de prueba del software resultante sea más alta [101].

4.3. Los lenguajes de modelado

Un modelo es una representación abstracta y más simple de alguna entidad real, en el caso de la ingeniería del software, del sistema software que se desea construir. De esta forma, con los diferentes modelos del software se busca representar diferentes características importantes a tener en cuenta de un sistema, con el objetivo de explicar y comprender la estructura y el comportamiento del software.

Por su parte, un lenguaje de modelado es un lenguaje artificial, formado por una serie de reglas, que permite describir la estructura de un sistema software, así como la información que maneja, a través de modelos. En el mundo del desarrollo de software existen diferentes lenguajes de modelado, los cuales pueden ser gráficos, que usan diagramas para representar conceptos y relaciones entre ellos, o textuales.

Dentro de los diferentes lenguajes de modelado y notaciones usadas en la construcción de modelos del software, como son las redes de Petri [293], máquinas de estados finitos (FSM) [361] o árboles de comportamiento (*behaviour trees*) [169], entre otros, destaca el uso del lenguaje de modelado UML. De hecho, algunas técnicas de pruebas desarrolladas en este trabajo usan este lenguaje de modelado para describir el sistema a probar, y generar automáticamente casos de prueba a partir de esta especificación (capítulos 6 y 7). Es por ello que en la siguiente sección se realizará una breve introducción a este lenguaje de modelado, explicando los conceptos básicos del mismo.

4.3.1. El lenguaje de modelado unificado UML

UML [126, 217, 315] es un lenguaje de modelado creado a mediados de los años noventa en Rational Software y adoptado como estándar en 1997 por el consorcio OMG [50], que lo ha respaldado desde dicha fecha. Aunque en muchas ocasiones se usa de manera informal, o sin aprovechar todas las características que ofrece [294], el lenguaje de modelado UML ha tenido una gran aceptación y acogida en la industria [214] y, en general, en la ingeniería del software [198].

El lenguaje UML ofrece elementos gráficos para modelar sistemas informáticos a través de diagramas y, de esta forma, especificar, visualizar, construir, y documentar diferentes aspectos de un sistema software como, por ejemplo, qué componentes forman parte de un sistema, cómo interaccionan entre ellos, cuales son las funcionalidades que ofrece un sistema, qué actores lo usan y cómo interaccionan con el mismo, las actividades que forman parte de un proceso realizado por el sistema, etc.

En general, los diagramas que proporciona UML permiten describir el sistema desde varios puntos de vista. Cada uno de estos diagramas tiene un propósito diferente, y permite describir al sistema desde una perspectiva distinta. Es posible agrupar los diferentes diagramas existentes en el lenguaje de modelado UML en diferentes *vistas* que permiten describir el sistema:

- **Vista del usuario,** que representa al sistema desde el punto de vista de los usuarios, conocidos como *actores* en el lenguaje UML. Los diagramas de casos de uso son el enfoque elegido para modelar esta vista.
- **Vista estructural,** en la que se usan diagramas en los que se representa la estructura estática de un sistema incluyendo clases, objetos, atributos, operaciones y relaciones entre ellos. Por ejemplo, los diagramas de clase describen la vista estructural de un sistema.
- **Vista del comportamiento,** para la que se utilizan diagramas que enfatizan el comportamiento dinámico de un sistema mostrando interacciones entre elementos estructurales, como son los objetos, y cambios en el estado interno de los mismos. Por ejemplo, los diagramas de secuencia, los diagramas de actividad o los diagramas de máquina de estados.
- **Vista de implementación,** donde, a través de los diagramas de componentes y de implementación, se representan cómo van a ser implementados los aspectos estructurales y de comportamiento.
- **Vista de despliegue,** que especifica cómo se distribuyen físicamente los diferentes componentes de un sistema, usando diagramas de despliegue.

El lenguaje de modelado UML se puede usar tanto para las tareas de análisis como para las de diseño. Así, mientras que el modelo de análisis se centra principalmente en las vistas del usuario y la vista estructural; el modelo de diseño se

dirige más a las vistas de comportamiento, implementación y despliegue.

4.3.2. El perfil de pruebas UML: *UML Testing Profile*

Existen una serie de extensiones específicas del lenguaje UML en diferentes áreas, las cuales se denominan perfiles [51]. Uno de estos perfiles, relacionado con las pruebas del software, es UTP [280], el cual ha sido creado para facilitar el diseño y desarrollo de pruebas del software usando el lenguaje de modelado UML.

Así, este perfil define conceptos para diseñar, visualizar, especificar, analizar, construir y documentar los componentes necesarios en las pruebas del software, y que no existen o son muy limitados en el propio lenguaje UML. En concreto, este perfil introduce los conceptos necesarios para definir la arquitectura de pruebas, el comportamiento de las mismas, los datos que se usarán en los casos de prueba, y el incluso modelar el concepto de "tiempo" en las pruebas (por ejemplo, para expresar restricciones de tiempo de respuesta en las llamadas a las operaciones del sistema a probar).

De esta forma, el perfil de pruebas UML permite el uso del lenguaje UML para la especificación de pruebas, cubriendo el hueco existente entre analistas y diseñadores con los programadores y los probadores de software, permitiendo una mejor colaboración entre todos ellos. Y así, proporciona soporte para aplicar la metodología de pruebas basadas en modelos usando el lenguaje de modelado UML [97].

Como se ha comentado anteriormente, el uso de modelos UML para la especificación del comportamiento del sistema a probar se usará en algunas aproximaciones de pruebas desarrolladas en este trabajo, en concreto, en la definición de las operaciones que forman parte de la interfaz de acceso a probar de un sistema software (capítulos 6 y 7).

4.4. Resumen

Un análisis y diseño adecuado, que se realice a partir de una especificación de requisitos de un sistema software, es el primer paso para la comenzar la implementación del mismo. Lograr un diseño que permita un mínimo acoplamiento entre los componentes que formarán parte del sistema, así como una máxima cohesión de los mismos, ayudará a aumentar la capacidad de prueba de un sistema software. Para ello, el uso de un lenguaje de modelado, como es UML, ayuda a realizar este tipo de tareas.

El caso de estudio empleado en este trabajo, es decir, el sistema VoDKATV, ha sido diseñado según estas directrices. Así, en el diagrama de componentes de la figura 3.6 de la página 56, que representa la arquitectura del sistema VoDKATV, se muestran una serie de componentes independientes, que se comunican entre sí a través de una serie de interfaces claras y definidas. Este hecho ha facilitado las

pruebas a nivel de componente e integración. Obviamente, en la implementación de cada componente individual se deben tener en cuenta estas mismas directrices para que las pruebas de unidad también pueden ser llevadas a cabo sin demasiadas complicaciones técnicas, como pueden ser las dependencias entre módulos.

5

Implementación de componentes individuales

5.1. Introducción

Dentro de los diferentes niveles de pruebas, las pruebas de unidad están diseñadas para que los programadores puedan comprobar el comportamiento de las unidades funcionales que implementen. A pesar de que todo el mundo acepta las ventajas de realizar este tipo de pruebas, y de llevarlas a cabo lo antes posible, en muchas ocasiones se suelen omitir, o no realizar con toda la rigurosidad necesaria.

Por un lado, muchas veces, este tipo de pruebas siguen desplazándose a las etapas finales de los proyectos, una vez que la implementación finaliza o está muy avanzada. No obstante, durante las últimas etapas de los proyectos, la presión por finalizar a tiempo suele aumentar, y si existen partes en las que se pueda recortar tiempo o recursos para cumplir con la planificación del proyecto, una de ellas suelen ser, de forma equivocada, las pruebas. Además, escribir las pruebas una vez que la implementación funciona, o bien bajo presión con intención demostrativa más que destructiva [194], puede conllevar a que existan funcionalidades que no se prueben, bien conscientemente o inconscientemente. Por otro lado, para muchos programadores las pruebas suelen ser un trabajo tedioso [227, 347], puesto que pueden requerir la preparación, ejecución y análisis de cientos o miles casos de prueba y, por ello, muchas veces no suelen prestarle toda la atención que se merecen [155].

En este capítulo se explica una metodología a seguir para realizar las pruebas de unidad, la cual intenta paliar las causas por las cuales éstas no suelen llevarse a

cabo de la mejor manera. Inicialmente, en la sección 5.2, se realiza una introducción a las pruebas de unidad, explicando las aproximaciones más habituales para llevarlas a cabo. Posteriormente, se describe cómo aplicar el desarrollo dirigido por las pruebas (sección 5.3) y las pruebas basadas en propiedades (sección 5.4) para realizar las pruebas de unidad. A continuación, en la sección 5.5, se explica, con un caso de estudio, cómo combinar estas dos aproximaciones para dar lugar a la nueva metodología desarrollada. Finalmente, en la sección 5.6 se resumen los contenidos de este capítulo.

5.2. Las pruebas de unidad

Una prueba de unidad [212, 213, 284] es una pieza de código escrita por un programador que invoca otra pieza de código que se quiere probar, normalmente una función, un método, una clase, un módulo o, en general, una "unidad" individual dentro del diseño de un sistema software que pueda ser probada de forma independiente; y comprueba si éste se comporta como se espera al ser ejecutado. De esta forma, en las pruebas de unidad, se realizan una serie de comprobaciones con un conjunto de casos de prueba. Así, se dice que la prueba de unidad falla si, como resultado de ejecutar el código a probar con alguno de los casos seleccionados, alguna de las comprobaciones realizadas no produce el resultado esperado.

Aunque todos los programadores han escrito algún tipo de prueba en algún momento para comprobar que su código fuente funciona como desean, las pruebas de unidad deben cumplir una serie de características. Así, las pruebas de unidad deben ser fáciles de implementar, legibles, mantenibles y ejecutarse rápidamente. También, es importante que estén totalmente automatizadas y sean repetibles, de tal forma que una vez escritas puedan ser usadas en el futuro, y no únicamente por el programador que las escribió, sino que cualquier otro programador debe ser capaz de poder ejecutarlas.

Para escribir las pruebas de unidad existen multitud de marcos de desarrollo [202] que, aunque no garantizan que las pruebas de unidad resultantes sean completas, legibles o mantenibles, ayudan al programador a realizar esta tarea. Entre ellos destacan las herramientas xUnit. Cabe destacar que xUnit no es una herramienta en sí misma, sino que es un término que se utiliza para referirse de manera genérica a todas aquellas herramientas de pruebas del software que están basadas en la herramienta SUnit [105, 106]. Por su parte, SUnit es un marco de trabajo creado por Kent Beck para organizar y ejecutar las pruebas de unidad para programas codificados en el lenguaje de programación Smalltalk. Con SUnit, las pruebas se realizan comparando valores obtenidos con valores esperados para determinar si la funcionalidad probada se ha implementado de la manera apropiada. Así, se automatiza la ejecución de las pruebas del software, permitiendo ejecutar un subconjunto o todas las pruebas definidas de forma cómoda.

Posteriormente, SUnit se portó a Java, creando el marco de trabajo JUnit [40, 212, 259, 341]. Actualmente, existen una gran variedad de marcos de desarrollo basados en SUnit y JUnit para diferentes lenguajes de programación. EUnit [26, 27, 133] para Erlang, NUnit [48, 213] para .NET, CUnit [17] para C, CppUnit [16] para C++, HUnit [32] para Haskell, PyUnit [56] para Python o QUnit [57] para JavaScript, son algunos ejemplos para los lenguajes de programación más comunes en la actualidad.

Este tipo de herramientas proporcionan un marco de trabajo que permite escribir las pruebas del software de una manera estructurada, ayudando al programador a través de una serie de utilidades y funciones predefinidas que se ofrecen en el propio lenguaje de programación para el que están diseñadas. También ofrecen una manera cómoda de ejecutar las pruebas, así como la revisión de los resultados de las mismas (número de casos ejecutados, casos fallidos y casos ejecutados exitosamente, etc.) bien de manera textual o a través de una interfaz gráfica. Normalmente, el código de pruebas se estructura en conjuntos de pruebas, de tal forma que cada conjunto de pruebas tiene asociadas una serie de acciones de inicialización, las propias pruebas que realizan las comprobaciones oportunas, y una serie de acciones de finalización que se ejecutan después de dichas pruebas. Estas pruebas, por su parte, se basan en el uso de aserciones para probar el código, y los casos de prueba a utilizar en la ejecución de las mismas son especificados manualmente en el propio código de pruebas.

Otro de los puntos importantes acerca de las pruebas de unidad trata sobre cuándo deberían escribirse. Muchos programadores escriben las pruebas de unidad después de implementar el software, siguiendo los pasos del diagrama de actividad mostrado en la figura 5.1; e, incluso, muchos consideran la actividad de escribir las pruebas como algo opcional. Sin embargo, las metodologías modernas, como el *desarrollo dirigido por las pruebas* [107], además de poner más énfasis en las tareas de pruebas, abogan por la implementación de éstas antes que el propio código fuente, eliminando así implícitamente su posible opcionalidad.

5.3. El desarrollo dirigido por las pruebas

El desarrollo dirigido por las pruebas (TDD) [95, 107, 218] es una técnica de diseño e implementación del software incluida dentro de la metodología XP. Su objetivo principal es producir *código limpio que funcione*, y se basa en tres pilares básicos:

- Implementar únicamente las funcionalidades que el cliente necesita.
- Minimizar los defectos del software que llegan a la fase de producción.
- Producir software modular, altamente reutilizable y preparado para el cambio.

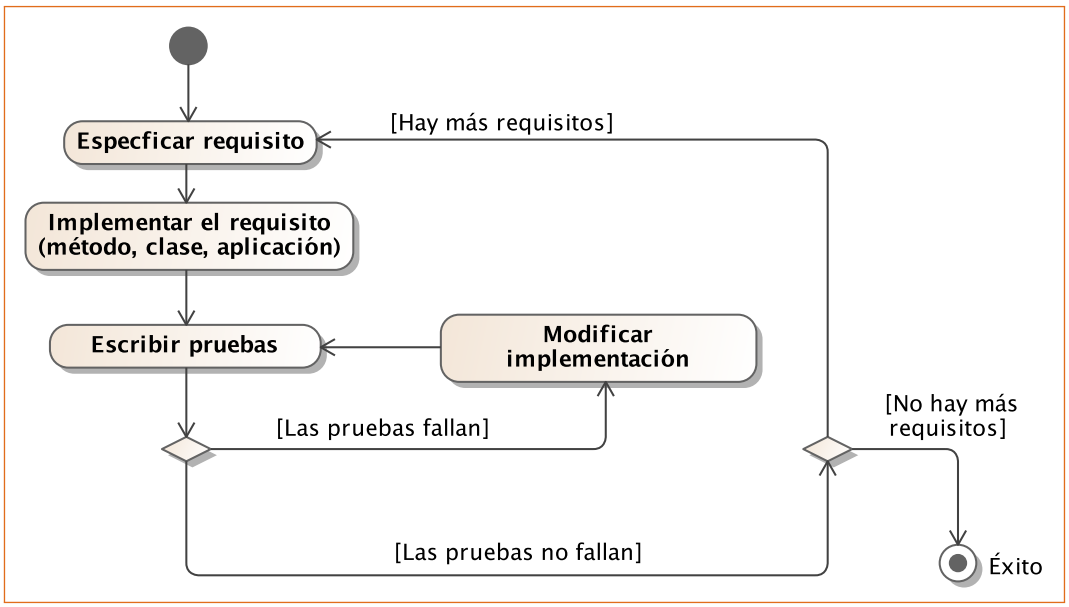


FIGURA 5.1: *Proceso tradicional para realizar las pruebas de unidad*

Para ello, esta metodología se centra en dirigir el desarrollo de software a través de la implementación de pruebas automáticas, para lo cual define los siguientes pasos a seguir (figura 5.2):

1. Elegir un requisito y escribir la especificación para el mismo. Para ello, se escribe una prueba que especifique cómo debe funcionar el sistema en relación a este requisito. Esta prueba es un ejemplo concreto de funcionamiento.

2. Ejecutar las pruebas existentes y comprobar que fallan (si el ejemplo anterior no falla, entonces es que ya había sido implementado, o es incorrecto). Para solucionarlo, implementar el código para que funcione con dicho ejemplo, y comprobar si las pruebas se ejecutan con éxito. El código resultante debería ser el mínimo necesario para que funcionen los ejemplos escritos, sin escribir más código del necesario.

3. Refactorizar [184], es decir, cambiar el código sin afectar a la funcionalidad, para eliminar duplicidad y hacer mejoras, tanto en la implementación como en el código de pruebas.

Estos pasos se repiten de manera iterativa a medida que se avanza en la implementación de cada componente. De esta forma, el código resultante se irá refinando y refactorizando en cada una de las iteraciones hasta lograr implementar la funcionalidad completa. Como se observa, las pruebas se programan antes que la propia implementación, lo cual evita el problema de que éstas no se realicen por falta de tiempo al finalizar del proyecto. Además, puesto que los programadores no "pue-

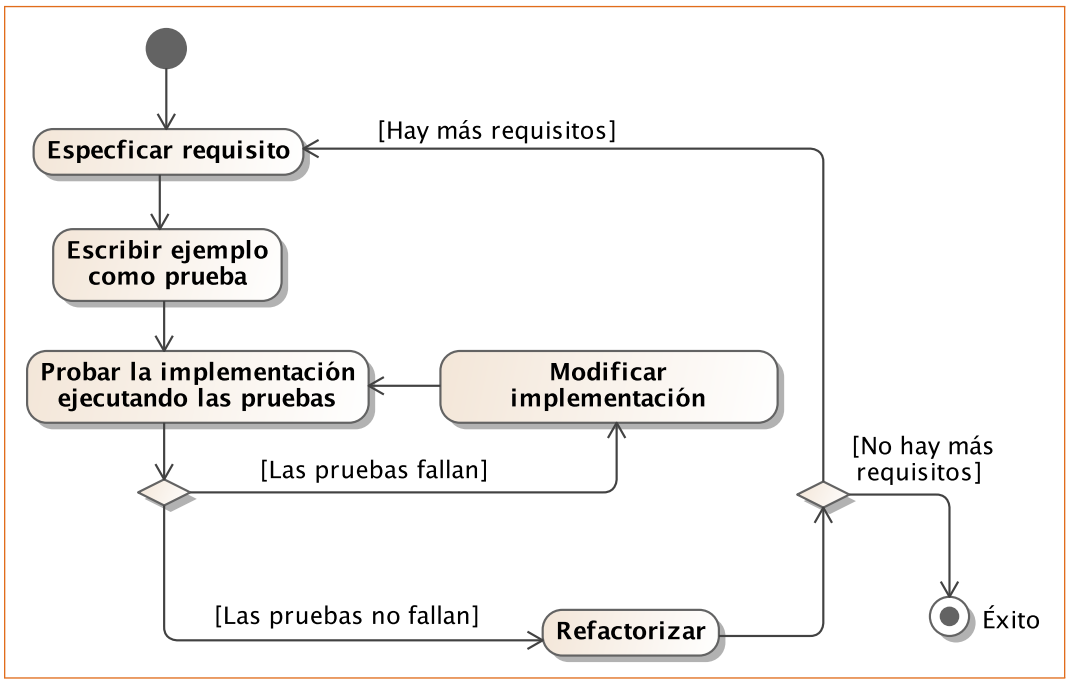


FIGURA 5.2: *Proceso a seguir en el desarrollo dirigido por las pruebas*

den" escribir código nuevo sin tener antes un caso de prueba que falle, es fácil conseguir una buena cobertura, e imposible escribir código sin probar.

Otra ventaja de esta aproximación es que, gracias a que únicamente se implementa el código necesario para pasar las pruebas, siendo éstas ejemplos que representan los requisitos a implementar, se evita escribir código innecesario, puesto que únicamente se debe escribir el mínimo código posible, y lo más simple posible. Este hecho provoca que el programador se centre en lo realmente importante. Así, a diferencia de las aproximaciones en las que las pruebas son escritas después del código, aquí el programador se debe centrar obligatoriamente en los requisitos antes de escribir el propio código.

Además, el hecho de que existan casos de prueba para el todo código implementado provoca que el programador confíe más en el código escrito, así como en futuras refactorizaciones necesarias en etapas posteriores de mantenimiento. Por otro lado, empezar escribiendo las pruebas provoca que el código resultante esté preparado para ser probado, algo que muchas veces no ocurre cuando las pruebas se piensan y escriben a posteriori.

En la actualidad, existen multitud de herramientas que permiten aplicar este método al desarrollo de software. En general, las herramientas xUnit, que se basan en el uso de aserciones para probar el código, son las que se suelen utilizar para implementar las pruebas. El uso de una interfaz gráfica que muestre el estado de las

pruebas con códigos de colores también suele ser habitual. De hecho, Beck [107] define las tres etapas de este método como *rojo*, que se refiere al período en el que se ha escrito la prueba, pero todavía falla; *verde*, cuando se ha escrito el código para que pasen las pruebas escritas; y *refactorización*, en la que se refactoriza el código resultante. En cualquier caso, cabe destacar que aunque el uso de las herramientas xUnit está ampliamente extendido en el mundo del software y cuando el desarrollo de software está dirigido por las pruebas son el tipo de herramientas más utilizadas, en este trabajo se propondrá el uso de las herramientas de pruebas basadas en propiedades en vez de este tipo de herramientas.

5.3.1. EUnit

A continuación se muestra un ejemplo de cómo se aplicaría el desarrollo dirigido por las pruebas para implementar, en el lenguaje de programación Erlang y utilizando la herramienta EUnit [26, 27, 133] (que es una herramienta xUnit para probar programas escritos en Erlang), la función `max` del módulo `lists_utils`. Esta función devuelve el máximo número entero de una lista no vacía de enteros:

```
lists_utils:max([integer()])-> integer()
```

El primer paso a realizar sería escribir una prueba simple que refleje, a modo de ejemplo, alguno de los requisitos más básicos, por ejemplo, que el máximo de una lista formada por el entero `0` es el propio entero `0`. Usando EUnit, este sería el primer caso de prueba:

```
max_test()-> ?assertEqual(0, lists_utils:max([0])).
```

Si se ejecuta, el resultado obtenido es, obviamente, que la función `max` del módulo `lists_utils` no existe, puesto que todavía no se ha implementado:

```
> eunit:test(lists_utils_test).
lists_utils_test:max_test (module 'lists_utils_test')... *failed*
::error:undef
 in function lists_utils:max/1
  called as max([0])
 in call from lists_utils_test:'-max_test/0-fun-0-'/1
 in call from lists_utils_test:max_test/0

=======================================================
 Failed: 1. Skipped: 0. Passed: 0.
error
```

Por tanto, se debe implementar una primera versión de la función `max` para que las pruebas escritas anteriormente se ejecuten exitosamente. En este caso, la implementación más simple y, evidentemente, no definitiva, sería la siguiente:

```
max(_L)-> 0.
```

De esta forma, al ejecutar las pruebas se obtendría un resultado positivo:

```
> eunit:test(lists_utils_test).
  Test passed.
ok
```

Evidentemente, los requisitos de la función `max` no quedan satisfechos sólo con este ejemplo, por lo que se deben añadir más ejemplos a las pruebas:

```
max_test()->
  ?assertEqual(0, lists_utils:max([0])),
  ?assertEqual(1, lists_utils:max([1])).
```

y comprobar si éstos siguen cumpliéndose con la implementación actual:

```
> eunit:test(lists_utils_test).
lists_utils_test:max_test (module 'lists_utils_test')... *failed*
::error:{assertEqual_failed,[{module,lists_utils_test},
                   {line,11},
                   {expression,
                    "lists_utils : max ( [ 1 ] )"},
                   {expected,1},
                   {value,0}]}
  in function lists_utils_test:'-max_test/0-fun-1-'/1

=======================================================
  Failed: 1. Skipped: 0. Passed: 0.
error
```

Como se observa, el resultado esperado como máximo de la lista `[1]` es `1`, mientras que la función `max` devuelve `0`. En este punto, es posible realizar una primera generalización de la función `max`, es posible suponer que el resultado es el primer elemento de la lista (lo cual es cierto cuando la lista tiene un único elemento):

```
max([H | _T])-> H.
```

De esta forma, se consigue que las pruebas se ejecuten con un resultado positivo:

```
> eunit:test(lists_utils_test).
  Test passed.
ok
```

En este punto, es necesario aumentar la funcionalidad de la función `max` para que permita calcular el máximo para listas de más de un elemento. Sin embargo,

antes de realizar esta tarea, como va a ser necesario añadir más casos de prueba, es posible pensar en refactorizar el código de pruebas para que esto sea más fácil, por ejemplo:

```
max_test()->
  TestCases = [{[0], 0}, {[1], 1}],
  [?assertEqual(Result, lists_utils:max(List)) ||
    {List, Result} <- TestCases].
```

De esta forma, añadir un nuevo caso de prueba implica simplemente añadir una nueva tupla en la lista `TestCases`. En este punto, si se vuelven a ejecutar las pruebas, el resultado sigue siendo el mismo:

```
> eunit:test(lists_utils_test).
  Test passed.
ok
```

Si ahora se añade un nuevo caso de prueba con una lista de dos elementos:

```
max_test()->
  TestCases = [{[0], 0}, {[1], 1}, {[0, 1], 1}],
  [?assertEqual(Result, lists_utils:max(List)) ||
     {List, Result} <- TestCases].
```

se comprueba que las pruebas dejan de funcionar, debido a que el resultado esperado para la lista `[0, 1]` es `1`, y no `0`, como devuelve la implementación actual de la función `max`:

```
> eunit:test(lists_utils_test).
lists_utils_test:max_test (module 'lists_utils_test')... *failed*
::error:{assertEqual_failed,[{module,lists_utils_test},
                  {line,22},
                  {expression,
                   "lists_utils : max ( List )"},
                  {expected,1},
                  {value,0}]}
  in function lists_utils_test:'-max_test/0-fun-0-'/2
  in call from lists_utils_test:'-max_test/0-fun-1-'/1
  in call from lists:all/2

=======================================================
  Failed: 1. Skipped: 0. Passed: 0.
error
```

Por tanto, este es el momento adecuado para añadir soporte para listas de dos elementos en la implementación de la función `max`:

```
max([H]) ->
 H;
max([H1, H2]) when H1 > H2->
 H1;
max([_H1, H2]) ->
 H2.
```

Con esta implementación, el resultado de las pruebas vuelve a ser satisfactorio:

```
> eunit:test(lists_utils_test).
 Test passed.
ok
```

Obviamente, se deben añadir más casos de prueba a los ya existentes, por ejemplo, listas de más de dos elementos:

```
max_test()->
 TestCases = [{[0], 0}, {[1], 1}, {[0, 1], 1},
     {[0, 1, 2], 2}],
 [?assertEqual(Result, lists_utils:max(List)) ||
    {List, Result} <- TestCases].
```

y, posteriormente, volver a ejecutar las pruebas, con lo que se obtiene el siguiente resultado:

```
> eunit:test(lists_utils_test).
lists_utils_test:max_test (module 'lists_utils_test')... *failed*
::error:function_clause
 in function lists_utils:max/1
  called as max([0,1,2])
 in call from lists_utils_test:'-max_test/0-fun-0-'/2
 in call from lists_utils_test:'-max_test/0-fun-1-'/1
 in call from lists:all/2

=======================================================
 Failed: 1. Skipped: 0. Passed: 0.
error
```

Como es de esperar, las pruebas ahora fallan porque la función `max` no está preparada la recibir listas de más de dos elementos, puesto que la implementación actual únicamente soporta listas de uno o dos elementos. Por tanto, es necesario generalizar la implementación de la función `max`:

```
max([H]) ->
   H;
max([H1, H2]) when H1 > H2->
   H1;
max([_H1, H2]) ->
   H2;
max([H1, H2 | T])->
   max([max([H1, H2]) | T]).
```

Así, cuando la función `max` recibe una lista de más de dos elementos, se calcula el máximo de los dos primeros elementos, y se vuelve a calcular de forma recursiva el máximo de la lista compuesta por el máximo resultante y el resto de elementos de la lista, hasta obtener una lista de dos elementos, en cuyo caso se aplicaría el mismo proceso ya implementado. Con esta implementación, las pruebas se vuelven a ejecutar con éxito:

```
> eunit:test(lists_utils_test).
  Test passed.
ok
```

Además, si se añaden más casos de prueba a los ejemplos existentes, el resultado seguirá siendo el mismo.

```
max_test()->
  TestCases = [{[0], 0}, {[1], 1}, {[0, 1], 1}, {[0, 1, 2], 2},
               {[-1, 10, 2], 10}, {[-1, 0, 0, 0], 0}],
  [?assertEqual(Result, lists_utils:max(List)) ||
     {List, Result} <- TestCases].
```

```
> eunit:test(lists_utils_test).
  Test passed.
ok
```

Como se ha comentado anteriormente, en este trabajo se propone utilizar el método de desarrollo dirigido por las pruebas, pero reemplazando el uso habitual de las herramientas xUnit, como ocurre en este ejemplo, por herramientas de pruebas basadas en propiedades.

5.4. Las pruebas basadas en propiedades

Las pruebas basadas en propiedades se pueden usar para realizar pruebas de unidad en componentes software. De hecho, este tipo de aproximaciones presentan grandes ventajas en comparación con las técnicas en las que los casos de prueba

son especificados manualmente. Así, en este caso, se usan propiedades para especificar el comportamiento del código a probar, es decir, cada uno de los requisitos a implementar. De esta forma, el programador escribe menos código de pruebas que en aproximaciones basadas en la especificación manual de los casos de prueba y, a la vez, se incrementa el número de casos de prueba utilizados, puesto que éstos son generados automáticamente a partir de las propiedades. Por tanto, el programador no debe centrarse en qué casos concretos usar para probar el código, ni pensar en si debe introducir o no más casos de prueba en función de cómo evolucione el código en el futuro.

El problema de elegir un buen conjunto de casos de prueba se transforma, por tanto, en generar una buena distribución de datos que cubra las propiedades definidas. Por ejemplo, si se usa QuickCheck como herramienta de pruebas basadas en propiedades es posible usar funciones como, por ejemplo, `collect` o `aggregate`, que permiten obtener información acerca de esta distribución de casos de prueba. Obviamente, también se puede realizar un análisis de cobertura del código a probar o del modelo de pruebas para evaluar los casos de prueba generados.

Es importante destacar que, aunque usando una aproximación basada en propiedades es posible utilizar modelos para describir el comportamiento del sistema a probar, en concreto, máquinas de estados, y, de hecho, su uso puede ayudar a encontrar defectos en algunas situaciones [87], este capítulo se centrará únicamente en el uso de propiedades independientes para realizar las pruebas de unidad, puesto que suelen ser suficientes en la mayoría de los casos. Sin embargo, si el uso de una máquina de estados es más adecuado, esta técnica también sería aplicable, para lo cual es posible seguir las indicaciones del capítulo 6.

Por ejemplo, utilizando la misma función `max` definida en la sección 5.3.1, que devuelve el máximo número entero de una lista no vacía de enteros:

```
lists_utils:max([integer()]) -> integer()
```

probar su funcionamiento con una aproximación basada en el uso de las herramientas xUnit (como se describió en la propia sección 5.3.1) significa escribir uno o varios casos de prueba que permitan comprobar si la función `max` se comporta como se espera, como podrían ser:

```
max([0]) == 0
max([1]) == 1
max([0, 1]) == 1
max([0, 1, 2]) == 2
...
```

Por el contrario, el enfoque usando pruebas basadas en propiedades consiste en escribir una o varias propiedades que describan el comportamiento esperado. Por ejemplo, la siguiente propiedad podría ser usada para describir el comportamiento

de la función `max`:

$$\forall L \subseteq \mathbb{Z},\ \ L \neq \emptyset \Rightarrow [(max(L) \geq I,\ \forall I \in L) \wedge max(L) \in L]$$

Esta propiedad indica que para toda lista de enteros L, si dicha lista no está vacía, entonces el máximo de dicha lista, calculado usando la función `max`, será un número entero mayor o igual que todos los elementos de la lista L y, además, dicho entero será un elemento de la lista.

De esta forma, esta propiedad, junto con el sistema a probar, es la entrada de la herramienta de pruebas. Esta herramienta, usando la propiedad especificada, genera cientos o miles de casos de prueba, en este caso, listas de enteros, y, para las listas no vacías, comprueba si la propiedad se cumple. La generación automática de miles de casos de prueba a partir de una propiedad de entrada ayuda a que se produzcan fallos que con una aproximación en la que se especifiquen los casos de prueba manualmente podrían permanecer ocultos.

Así, si la implementación de la función `max` no fuese correcta, por ejemplo, si las comparaciones de valores se realizasen usando los valores absolutos de los mismos en vez de los valores reales:

```
max([H])->
   H;
max([H1, H2]) when abs(H1) > abs(H2)->
   H1;
max([_H1, H2]) ->
   H2;
max([H1, H2 | T])->
   max([max([H1, H2]) | T]).
```

el resultado devuelto no sería el esperado cuando la lista que se utiliza como entrada contiene números negativos, en concreto, cuando la lista tiene más de un elemento y alguno de ellos es negativo.

En este caso se necesitaría al menos un caso de prueba con una lista de estas características que permitiese comprobar este comportamiento. Si el programador no recuerda o no decide introducir un caso de prueba como este en su conjunto de pruebas, este problema no se detectaría. Sin embargo, y aunque depende en gran medida de la distribución utilizada para generar los datos de entrada que se usarán como casos de prueba a partir de la especificación de entrada, la cual se basa en algún tipo de distribución estadística aleatoria, es muy probable que la herramienta de pruebas basadas en propiedades detecte este defecto con la propiedad introducida.

5.4.1. QuickCheck

Utilizando la herramienta QuickCheck es posible implementar la propiedad formulada anteriormente para describir el comportamiento de la función `max`, la cual se escribiría de la siguiente manera:

```
prop_max() ->
 ?FORALL(L, eqc_gen:list(eqc_gen:int()),
  ?IMPLIES(length(L) > 0,
   begin
    Result = max(L),
    lists:all(fun(I) -> Result >= I end, L) andalso
       lists:member(Result, L)
   end)).
```

`FORALL` es una macro Erlang que proporciona QuickCheck para comprobar que una propiedad se cumple para todos los elementos generados; `?IMPLIES` es otra macro usada para descartar casos que no cumplen una determinada condición; e `int()` y `list(int())` son generadores de enteros y listas de enteros respectivamente, ambos ya incluidos con la librería de generadores básicos de QuickCheck.

A partir de esta especificación, la herramienta QuickCheck es capaz de generar automáticamente casos de prueba, ejecutarlos, y comprobar el resultado obtenido para cada uno de ellos. Por defecto, QuickCheck genera 100 casos de prueba utilizando los generadores de datos, en este caso, `list(int())`, que generará listas de números enteros, pero es posible especificar un número de casos de prueba a generar (usando la función proporcionada por QuickCheck `numtests`).

En general, las distribuciones de datos usadas por QuickCheck en sus propios generadores suelen funcionar de tal manera que los primeros datos que se generan suelen ser los más "pequeños", siendo más "grandes" progresivamente. Por ejemplo, en el caso del generador de enteros, `int()`, los primeros enteros generados estarán próximos a 0, y a medida que se generen más, la probabilidad de que se genere un entero más grande, tanto positivo como negativo, será cada vez mayor. Con respecto al generador `list(int())`, se comenzarán generando la listas vacías y listas pequeñas, y, progresivamente, se generarán listas con más elementos.

En este caso, el resultado de ejecutar la propiedad con QuickCheck para probar la implementación incorrecta de la función `max` (la cual no funcionaba de la manera esperada con números enteros negativos) es el siguiente:

```
> eqc:quickcheck(lists_utils:prop()).
xxxxxxxxxx.x.xxxx.xx.x.xx.xx...x.......x
Failed! After 17 tests.
[-5,-5,-1]
```

La herramienta QuickCheck muestra el carácter `"x"` cuando un caso generado es descartado porque no se cumple la condición especificada con `?IMPLIES`, y el carácter `"."` cuando el caso generado es realmente usado por la propiedad. En concreto, los casos de prueba generados hasta encontrar uno que no cumple la propiedad definida han sido los siguientes: `[-1]`, `[-1]`, `[0]`, `[2]`, `[2]`, `[-1]`, `[-3]`, `[2]`, `[-4]`, `[5,0]`, `[2]`, `[3]`, `[4]`, `[-5]`, `[-1]`, `[1,1]`, `[-5,-5,-1]`. Como se observa, se comienzan generando listas de 1 elemento y, progresivamente, se generan listas de más elementos y con números enteros cada vez más "grandes" (tanto positivos como negativos). Por otro lado, el decimoséptimo caso de prueba generado, `[-5,-5,-1]`, no cumple la propiedad especificada. De esta forma, es posible encontrar el defecto con números enteros negativos sin necesidad de confiar en que el programador haya recordado añadir un caso de prueba similar en sus pruebas en las que los casos de prueba se especifican manualmente.

Una de las ventajas del uso de QuickCheck es que cuando éste encuentra un contraejemplo, como ocurre en este caso, QuickCheck aplica un proceso de reducción y simplificación (conocido con el término inglés *shrinking*) mediante el cual se intenta obtener el mínimo caso de prueba equivalente al caso de prueba original que ha causado el fallo [87, 135, 374]. Conocer este caso de prueba mínimo suele facilitar la búsqueda de la raíz de la causa del fallo producido, puesto que al mostrar contraejemplos más simples, este proceso de reducción ayuda al programador a detectar por qué la propiedad ha fallado. Aunque el comportamiento de este proceso es personalizable por el programador, por defecto, QuickCheck se basará en el contraejemplo encontrado para generar casos cada vez más "pequeños", con el objetivo de buscar otro caso de prueba más simple que cause un fallo en el sistema. En este sentido, el proceso de reducción contrasta con el proceso de generación, en el que se generan valores cada vez más "grandes".

Así, en el ejemplo de la función `max`, después de aplicar el proceso de reducción, QuickCheck devuelve el siguiente caso como contraejemplo, el cual está formado por una lista de dos elementos, uno de ellos negativo:

```
Shrinking..(2 times)
[0,-1]
```

En este caso, una lista formada por dos elementos con uno de ellos negativos (el entero `-1`), es el "mínimo" caso de prueba devuelto por QuickCheck que provoca que la especificación de la función `max` no se cumpla, puesto que el resultado obtenido es `-1` en vez de `0`. Un contraejemplo como este puede dar alguna pista al programador de que el fallo podría estar relacionado con la inclusión de números negativos en la lista de entrada.

Finalmente, destacar que, además de `FORALL` e `IMPLIES`, QuickCheck proporciona de una serie de macros Erlang como `ALWAYS`, `SOMETIMES`, o `ONCEONLY`, entre muchas otras, que ayudan a escribir propiedades. Por otra parte, QuickCheck,

además de proporcionar una serie de generadores de datos predefinidos (números enteros, números reales, números naturales, caracteres, listas, etc.), soporta la construcción de generadores propios para tipos de datos complejos. La habilidad de usar los generadores que proporciona QuickCheck directamente, o usarlos como base para construir generadores personalizados o más complejos, suponen una mejora significativa con respecto a las pruebas tradicionales [295], en las que normalmente los valores usados los elige manualmente la persona que escribe dichas pruebas. En consecuencia, cuando se usa una aproximación basada en propiedades, la calidad de los casos de prueba depende en gran medida de la calidad de los generadores de datos usados.

5.5. El desarrollo dirigido por propiedades

Llevar a la práctica el desarrollo dirigido por las pruebas como metodología de desarrollo de software conlleva numerosas ventajas con respecto a las metodologías clásicas en las que las pruebas se escriben al final del desarrollo. Código de mayor calidad, más fiable, robusto e incluso mejor estructurado y reutilizable son algunas de estas ventajas. No obstante, para que esto ocurra, los casos de prueba elegidos como ejemplos de funcionamiento deben ser lo suficientemente representativos, además de contener casos límite donde los fallos son más propensos a suceder. Por tanto, el programador debe elegir un buen conjunto de casos de prueba y, para ello, debe tener un amplio conocimiento de la implementación para que, de esta forma, la cobertura de las pruebas sea alta. Por ejemplo, la no inclusión de un caso de prueba que permita comprobar el funcionamiento de la función `max`, descrita en la sección 5.4, con listas que contienen números enteros negativos no permitiría nunca encontrar el defecto descrito en dicha sección.

Este problema puede ser minimizado si se usa una técnica de generación automática de casos de prueba como son las pruebas basadas en propiedades. De esta forma, se escriben propiedades que describen el comportamiento a probar en vez de casos de prueba concretos. Es por ello que en este capítulo se propone una combinación de estas dos técnicas, es decir, el desarrollo dirigido por las pruebas y las pruebas basadas en propiedades, dando lugar a una nueva metodología, a la que este trabajo se referirá con el nombre de *desarrollo dirigido por las pruebas* (PDD) [311].

Con esta nueva metodología, el problema de la elección de un buen conjunto de casos de prueba se transforma en generar una buena distribución de los datos de entrada para cubrir las propiedades definidas. Además, puesto que las pruebas basadas en propiedades permiten incrementar la cobertura de las pruebas sin aumentar proporcionalmente el esfuerzo dedicado a escribirlas, esta aproximación no aumenta la complejidad de la metodología original.

El término PDD ya ha sido utilizado otros trabajos, sin embargo, la connotación no es la misma que la dada aquí. Por ejemplo, [102] describe una técnica en la que se desarrollan conjuntamente pruebas automáticas, especificaciones formales y modelos UML ejecutables en vez de realizar uno después de otro, usando OCL para expresar propiedades. En el presente trabajo, la idea es diferente, puesto que el término PDD (*Property-Driven Development*) se usa para referirse a una combinación de dos metodologías existentes: el desarrollo dirigido por las pruebas (*Test-Driven Development*) y las pruebas basadas en propiedades (*Property-Based Testing*).

5.5.1. Metodología de pruebas

La metodología propuesta para llevar a cabo el desarrollo dirigido por propiedades es similar a la existente para el desarrollo dirigido por las pruebas, salvo las implicaciones resultantes de escribir propiedades en vez de ejemplos concretos para probar el código resultante (ver figura 5.3). Así, si se comparan ambos procesos, las dos aproximaciones comienzan el desarrollo escribiendo las pruebas antes que la propia implementación. Sin embargo, a diferencia del desarrollo dirigido por las pruebas, donde el programador escribe ejemplos concretos, en el desarrollo dirigido por propiedades las pruebas se escriben en forma de propiedades que representan los requisitos del software a implementar. A partir de las propiedades escritas, las herramientas de pruebas se encargan de generar cientos o miles de casos de prueba automáticamente.

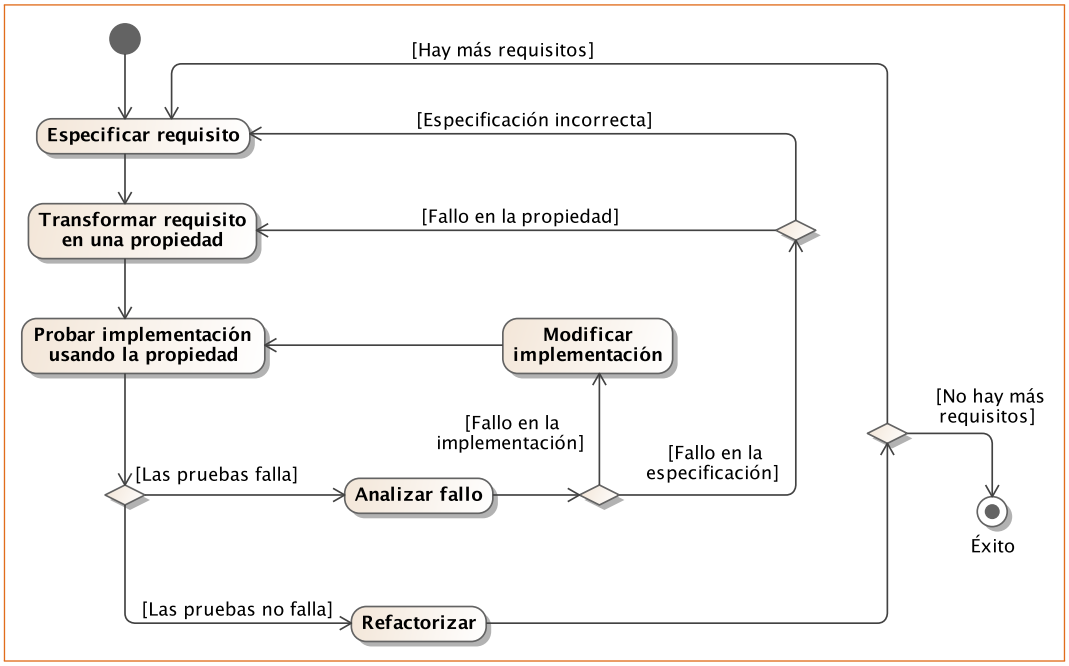


FIGURA 5.3: *Proceso a seguir en el desarrollo dirigido por propiedades*

Una de las implicaciones de usar propiedades en vez de casos de prueba concretos es que éstas son más propensas a errores que los ejemplos usados en el desarrollo dirigido por las pruebas, puesto que son abstracciones y no ejemplos concretos. Es por ello que cuando se encuentra un fallo probando la implementación, se debe analizar si dicho fallo proviene de un defecto de la propia implementación, o bien de la propiedad escrita para probar dicha implementación.

En cualquier caso, una vez conseguido que las pruebas se ejecuten exitosamente con la implementación, los pasos a seguir serían exactamente los mismos que en las pruebas basadas en propiedades, es decir, refactorizar el código resultante y repetir el proceso en caso de que haya más requisitos.

5.5.2. Caso de estudio: implementación de una librería de plantillas

Esta sección ilustra cómo se ha aplicado el desarrollo dirigido por propiedades a un caso de estudio real [311]. En concreto, dicho caso de estudio es una librería de plantillas que se ha implementado en el lenguaje de programación Erlang. Esta librería se encarga de realizar substituciones en cadenas de caracteres que se encuentren entre los símbolos `@`. Por ejemplo, `"Hola @nombre@"` es una plantilla con una variable `"nombre"`, que, si se substituye por `"mundo"`, se obtendría como resultado `"Hola mundo"`.

La función que realiza las substituciones, cuya implementación se ha realizado según un enfoque dirigido por propiedades, tiene como nombre `string` y está definida en el módulo Erlang `lstd_template`, teniendo la siguiente especificación de tipos:

```
lstd_template:string(string(), [{string(), string()}])->
   string()
```

donde el primer parámetro es la cadena de caracteres con las variables a substituir (que se llamará *plantilla*), y el segundo parámetro es una lista de tuplas con cada una de las substituciones a realizar.

Si se siguiese un desarrollo dirigido por las pruebas, el primer paso sería añadir un ejemplo trivial que requiera una implementación también trivial, e ir avanzando con la implementación de la función `string` añadiendo progresivamente más ejemplos a las pruebas. La selección de los ejemplos usados como casos de prueba condiciona la calidad de las pruebas con las que el código resultante será probado. Por ejemplo, aunque no como primer caso de prueba puesto que sería demasiado complejo para comenzar, el programador podría escribir ejemplos como el siguiente:

```
?assertEqual("hello world",
 lstd_template:string("hello @name@", [{"name", "world"}])).
```

Además, es muy probable que el programador también quiera incluir ejemplos de funcionamiento para implementar el comportamiento especial del carácter `@`:

```
?assertEqual("@", lstd_template:string("@@", [])).
```

Y, por supuesto, sería interesante probar el comportamiento en casos límite o poco comunes, como son textos que contentan el carácter `@`, cadenas vacías `""` o textos con saltos de línea `\n`, entre otros:

```
?assertEqual("foo@bar.com",
   lstd_template:string("@name@@@bar.com",
    [{"name", "foo"}])).

?assertEqual("foo@bar.com",
   lstd_template:string("@name@@@@domain@",
    [{"name", "foo"}, {"domain", "bar.com"}])).

?assertEqual("",
   lstd_template:string("@var@",[{"var", ""}])).

?assertEqual("Text\nwith lines",
   lstd_template:string("@var@\nwith lines",
    [{"var", "Text"}])).
```

Como se observa, incluso para módulos simples es difícil cubrir todos los casos. Textos con el carácter `@`, cadenas vacías `""` o textos con saltos de línea `\n`, son algunos de los casos límite que un programador debería incluir como parte de las pruebas de la función `string`. Sin embargo, en muchas ocasiones, existen casos límite olvidados en los ejemplos especificados, o incluso éstos podrían ir cambiando a medida que evoluciona la implementación.

Cuando el desarrollo se dirige por propiedades, el programador debe pensar en propiedades en vez de casos de prueba concretos y, siguiendo la misma filosofía que en el desarrollo dirigido por las pruebas, se debe comenzar escribiendo propiedades sencillas que permitan ir implementando el código fuente de manera iterativa. Por ejemplo, una primera propiedad podría ser que *una lista de substitución vacía debe dejar la plantilla intacta*:

```
prop_string_empty_list()->
  ?FORALL(S, ql_gen:string(),
     S =:= lstd_template:string(S, [])).
```

donde el generador `string` del módulo `ql_gen` genera una lista de caracteres imprimibles (códigos ASCII dentro de los rangos 8 a 13, 32 a 126, y 27).

Al ejecutar la propiedad con QuickCheck, el primer error encontrado es que no

existe la función `string` en el módulo `lstd_template`. Por tanto, el primer paso es escribir una primera implementación de dicha función que cumpla la especificación dada:

```
string(String, _Subs)-> String.
```

Evidentemente, esta implementación no será la implementación final, pero se refinará de forma progresiva. De la misma manera, al igual que el código de la función a implementar, es muy común que las propiedades escritas tampoco sean correctas en su primera implementación. De hecho, la implementación actual de la propiedad `prop_string_empty_list` es incorrecta, y fallará después de implementar algunos detalles de la librería. En cualquier caso, comenzar con implementaciones simples asegura que tanto la implementación como las pruebas evolucionen en paralelo, y que el nuevo comportamiento introducido en el código sea realmente probado con las propiedades escritas.

El siguiente paso en la implementación es pensar en cómo debería funcionar el proceso de substitución y, de esta forma, definir más propiedades para continuar con la implementación. Una posible propiedad podría ser que los literales que estén entre caracteres especiales, en este caso, `@`, es decir, las variables de las plantillas, se substituyan por otros valores. Por ejemplo, si en la plantilla `"Hola @nombre@"` se substituye la variable `nombre` por otro la valor, entonces se obtendría la cadena de caracteres `"Hola "` con dicho valor al final.

Sin embargo, definir esta propiedad para implementarla es todavía un paso muy complejo en este punto, por lo que en estos casos lo que se debe realizar es escribir una combinación de propiedades más simples para poder llevar a cabo la implementación de forma iterativa. En concreto, el proceso de implementación de la librería de plantillas se ha dividido en los siguientes pasos:

- Implementar un *scanner*.
- Implementar un *parser*.
- Implementar el proceso de substitución.
- Añadir soporte para el carácter `@`, el cual se considerará como un carácter especial.

Por último, se analizarán los generadores usados.

5.5.2.1. Implementación del *scanner*

Primeramente, es obvio que la función `string` que se va a implementar debe ser capaz de detectar los caracteres `@` para realizar las substituciones. Por lo tanto, se necesitará un *scanner* que devuelva los *tokens* de una cadena de caracteres de

entrada. Estos *tokens* serán, o bien literales (`{string, string()}`), o bien el carácter `@` (`at`):

```
lstd_template:tokens(string()) -> [{string, string()} | at]
```

Siguiendo la metodología propuesta, el primer paso para implementar el *scanner* es escribir propiedades que permitan comprobar el comportamiento del mismo, en concreto, examinando el resultado obtenido a partir de una serie de entradas generadas. Para ello, una técnica habitual es generar una *representación* de datos de entrada con cada salida correspondiente esperada, además de implementar funciones para mostrar estas representaciones de la entrada y la salida. En este caso, se definen los siguientes tipos de datos que manejará el *scanner*:

```
template() = [var() | text()]
var() = {var, string()}
text() = {text, string()}
```

donde `template()` es una representación de una plantilla, que está formada por variables (`var()`) y secciones de texto (`text()`). Por ejemplo, con esta definición de tipos de datos, la representación para la plantilla `"Hola @nombre@"` es la lista de tuplas:

```
[{text,"Hola "},{var,"nombre"}]
```

Además, es posible implementar un generador de datos QuickCheck que genere este tipo de representaciones de la siguiente forma:

```
template()->
  eqc_gen:list(eqc_gen:oneof(
     [{var, ql_gen:string()}, {text, ql_gen:string()}])).
```

Por otro lado, también es interesante implementar funciones que a partir de una representación sean capaces de generar la cadena de caracteres que ésta representa:

```
> to_string([{text,"Hola "},{var,"nombre"}]).
"Hola @nombre@"
```

así como la lista esperada de *tokens*:

```
> to_tokens([{text,"Hola "},{var,"nombre"}]).
[{string,"Hola "},at,{string,"nombre"},at]
```

La implementación de las funciones `to_string` y `to_tokens` es la que se muestra a continuación:

```
to_string(Template)->
 lists:concat([to_string_acc(X) || X<-Template]).

to_string_acc({var, V})->
 lstd_string:format("@~s@", [V]);
to_string_acc({text, S})->
 S.
```

```
to_tokens(Template)->
 lists:concat([to_tokens_acc(X) || X<-Template]).

to_tokens_acc({var, S})->
 [at, {string, S}, at];
to_tokens_acc({text, S})->
 [{string, S}].
```

Una vez definidos los tipos de datos involucrados y su correspondiente generador de datos, así como las funciones auxiliares `to_string` y `to_tokens`, ya es posible escribir la primera propiedad para definir el *scanner*: *para toda representación de entrada, la función tokens devuelve la lista esperada de tokens*:

```
prop_tokens() ->
 ?FORALL(T, template(),
  to_tokens(T) == lstd_template:tokens(to_string(T))).
```

Puesto que todavía no existe la función `tokens` implementada en el módulo `lstd_template`, es necesario escribir una primera implementación para que la propiedad pueda ser ejecutada:

```
tokens([]) ->
 [];
tokens([$@ | T]) ->
 [at | tokens(T)];
tokens(S) ->
 {String, T} = lists:split(string:cspan(S, "@"), S),
 [{string, String} | tokens(T)].
```

A simple vista, esta implementación podría parecer que es correcta y que cumple con la especificación, es decir, con la propiedad definida, incluso si se valida manualmente con algún caso de prueba. Sin embargo, al ejecutar la propiedad, QuickCheck encuentra varios errores. El primero de los errores encontrados es el siguiente:

```
> eqc:quickcheck(lstd_template_eqc:prop_tokens()).
..Failed! After 3 tests.
[{text,[]}]
false
```

el cual se debe a un caso límite con literales vacíos:

```
> lstd_template_eqc:to_tokens([{text,""}]).
[{string,[]}]

> lstd_template_eqc:to_string([{text,""}]).
[]

> lstd_template:tokens("").
[]
```

Este es un defecto en la especificación, puesto que si el generador puede crear representaciones con literales vacíos, entonces es imposible parsear la representación de una forma determinista. Esto se soluciona fácilmente usando un generador de datos que no genere cadenas de caracteres vacías para los literales:

```
template()->
  eqc_gen:list(eqc_gen:oneof(
     [{var, ql_gen:string()},
      {text, non_empty_string()}])).
```

Después de este cambio, QuickCheck todavía es capaz de encontrar más errores:

```
> eqc:quickcheck(lstd_template_eqc:prop_tokens()).
.......Failed! After 8 tests.
[{var,[]}]
```

lo cual parece ser otro caso límite:

```
> lstd_template_eqc:to_tokens([{var,""}]).
[at,{string,[]},at]

> lstd_template_eqc:to_string([{var,""}]).
"@@"

> lstd_template:tokens("@@").
[at,at]
```

El problema, en este caso, es que el *scanner* no tiene en cuenta que entre dos caracteres `@` debe haber un nombre de variable. La propiedad usada en la especificación, sin embargo, introduce un nombre de variable vacío: `{var,""}`. Como

los nombres de variables vacíos no aportan valor y, en cambio, complican enormemente la implementación y las pruebas, finalmente se modifica el generador para no permitir nombres de variables vacíos:

```
template()->
 eqc_gen:list(eqc_gen:oneof(
   [{var, non_empty_string()}, {text, non_empty_string()}])).
```

Después de ejecutar la propiedad de nuevo, QuickCheck encuentra un nuevo error:

```
> eqc:quickcheck(lstd_template_eqc:prop_tokens()).
..............Failed! After 25 tests.
[{text,"gk$"},{text,"T\">m"}]
Shrinking....(4 times)
[{text," "},{text," "}]
false
```

el cual se corresponde con otro defecto en la especificación, puesto que no es posible diferenciar entre dos secciones de literales consecutivas y una sección única con la concatenación de sus textos:

```
> lstd_template_eqc:to_tokens(
   [{text," "},{text," "}]).
[{string," "},{string," "}]

> lstd_template:tokens(" ").
[{string," "}]
```

Para solucionar este problema se añade un postprocesado de los literales consecutivos en el generador de plantillas. Para ello, se usa la macro `LET` proporcionada por QuickCheck, que permite realizar transformaciones de un valor previamente generado, en este caso, una representación de la plantilla, la cual se transformará concatenando las secciones de literales consecutivas:

```
template()->
 ?LET(L, eqc_gen:list(
    {eqc_gen:oneof([text,var]), non_empty_string()}),
    fold_text(L)).

fold_text([{text, A}, {text, B} | T])->
 fold_text([{text, A++B} | T]);
fold_text([H | T])->
 [H | fold_text(T)];
fold_text([])->
 [].
```

Sin embargo, una vez más, la propiedad sigue sin funcionar de la manera esperada:

```
> eqc:quickcheck(lstd_template_eqc:prop_tokens()).
..............Failed! After 25 tests.
[{var,"~"},{var,"<0)"},{var,"@b39"}]
Shrinking..(2 times)
[{var,"@"}]
false
```

En este caso, QuickCheck ha encontrado un error con una plantilla compleja, aunque después de aplicar el proceso de reducción con el contraejemplo encontrado, el problema parece obvio. Es otro defecto en la especificación, puesto que el carácter `@` no puede ser usado como nombre de variable, ni en los literales (QuickCheck encontrará el mismo error en una sección de texto después de ejecutar la propiedad varias veces más). Para solucionar esto, se cambia el generador de plantillas para evitar generar caracteres `@`, usando la siguiente implementación del generador `valid_string`:

```
valid_char() ->
 ?SUCHTHAT(C, ql_gen:printable(), C =/= $@).

valid_string() ->
 ql_gen:non_empty_list(valid_char()).
```

Por lo tanto, la implementación del generador de plantillas quedaría de la siguiente manera:

```
template() ->
 ?LET(L, eqc_gen:list(
    {eqc_gen:oneof([text,var]), valid_string()}),
    fold_text(L)).
```

Después de este último paso, QuickCheck no es capaz de encontrar ningún otro contraejemplo. En este punto, las pruebas están lo suficientemente refinadas como para cubrir un gran número de casos límite, y el *scanner* funciona acorde a su especificación. Se debe tener en cuenta que la especificación prohíbe explícitamente el uso de `@` como texto, aunque esto será tratado más adelante.

5.5.2.2. Implementación del *parser*

Una vez que el *scanner* está funcionando, el siguiente paso es especificar el funcionamiento del *parser*. En este caso, la única responsabilidad del *parser* es localizar los caracteres `@` para dividir la entrada en secciones de texto y variables. Para ello, se implementará una función `parse` con la siguiente especificación de tipos:

```
lstd_template:parse([{string, string()} | at]) -> template()
```

La misma representación usada para las plantillas hasta ahora puede ser también utilizada como salida esperada del *parser*, ya que aparentemente tiene toda la información del proceso de substitución de variables que se necesita. Un ejemplo de funcionamiento del *parser* es el siguiente:

```
> lstd_template:parse("Hola @nombre@").
[{text,"Hola "},{var,"nombre"}]
```

De esta forma, es posible escribir la siguiente propiedad QuickCheck que permite especificar el funcionamiento del *parser*:

```
prop_parse() ->
  ?FORALL(T, template(),
     T == lstd_template:parse(
        lstd_template:tokens(to_string(T)))).
```

la cual comprueba que aplicar la función `parse` a los *tokens* obtenidos para la representación de cualquier plantilla generada, produce como resultado la misma representación de plantilla original.

La siguiente implementación se comporta de la manera esperada para todos los casos de prueba generados por QuickCheck a partir de la propiedad anterior:

```
parse(Tokens)->
  parse(Tokens, text).

parse([at | T], Terminal)->
  parse(T, switch_terminal(Terminal));
parse([{string, S} | T], Terminal)->
  [{Terminal, S} | parse(T, Terminal)];
parse([], _)->
  [].

switch_terminal(text)->
  var;
switch_terminal(var)->
  text.
```

Por tanto, en este punto del desarrollo, hay una implementación fiable que transforma una cadena de caracteres en una representación (elegida para facilitar las pruebas), y viceversa. Además, esta implementación tiene asociada propiedades, las cuales pueden usarse tanto para la especificación de las pruebas como para documentación.

5.5.2.3. Implementación del proceso de substitución

El siguiente paso en la implementación es especificar cómo se realizará el proceso de substitución de variables por su valor, lo cual, evidentemente, también debe ser probado. Para ello, la representación elegida para las plantillas debe ser enriquecida con más información, puesto que en la representación actual existen las variables, pero no el valor asignado a ellas. Teniendo esto en cuenta, esta es la siguiente versión del generador de plantillas:

```
template()->
 ?LET(L, eqc_gen:list(eqc_gen:oneof([text(),var()])),
    fold_text(L)).

text()->
 {text, valid_string()}.

var()->
 {var, valid_string(), ql_gen:string()}.
```

También se debe cambiar la implementación de las funciones auxiliares `to_string` y `to_tokens`:

```
to_string_acc({var, V, _})->
 lstd_string:format("@~s@", [V]);
to_string_acc({text, S}) ->
 S.
```

```
to_tokens_acc({var, S, _})->
 [at, {string, S}, at];
to_tokens_acc({text, S})->
 [{string, S}].
```

En este punto, si se ejecutan las propiedades de nuevo, se puede observar como `prop_parse` ya no es una propiedad válida, puesto que la representación de la plantilla y el resultado del proceso de parseado ya no son equivalentes. Es por ello que se debe cambiar la implementación de esta propiedad:

```
prop_parse()->
 ?FORALL(T, template(),
    to_parsed(T) == lstd_template:parse(
       lstd_template:tokens(to_string(T)))).

to_parsed(Template)->
 [to_parsed_acc(X) || X<-Template].
```

```
to_parsed_acc({var, Name, _Value})->
 {var, Name};
to_parsed_acc({text, S})->
 {text, S}.
```

Por otro lado, junto con la función auxiliar `to_string`, definida anteriormente, se implementarán dos nuevas funciones que ayudarán a formular la propiedad que permitirá probar la función `string`. Estas dos nuevas funciones son `to_substs` y `to_result`, cuyo funcionamiento (junto con la función `to_string`) se muestra en la figura 5.4 con un ejemplo práctico.

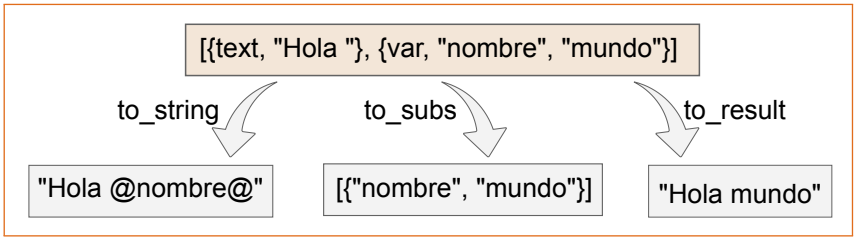


FIGURA 5.4: *Funciones `to_string`, `to_substs` y `to_result`*

Estas dos funciones se han implementado de la siguiente manera:

```
to_substs([])->
 [];
to_substs([{var, Name, Value} | T])->
 [{Name,Value} | to_substs(T)];
to_substs([{text,_} | T])->
 to_substs(T).
```

```
to_result(Template)->
 lists:concat([to_result_acc(X) || X<-Template]).

to_result_acc({var, _, S})-> S;
to_result_acc({text, S})-> S.
```

En este momento ya es posible formular la propiedad que define los casos positivos para la función `string`:

```
prop_string() ->
  ?FORALL(T, template(),
     ?LET({Substs, String, Expected},
        {to_substs(T), to_string(T), to_result(T)},
        ?LET(Result, lstd_template:string(String, Substs),
           ?WHENFAIL(
              io:format(
                 "~nTemplate: ~p~n"
                 "Substs : ~p~n"
                 "Expected: ~p~n"
                 "Got : ~p~n",
                 [String, Substs, Expected, Result]),
              Expected =:= Result)))).
```

En esta propiedad se ha añadido código para facilitar el depurado, utilizando para ello la macro `WHENFAIL` de QuickCheck. Naturalmente, la implementación inicial de `string`, la cual simplemente devolvía el mismo valor que recibe como entrada, no cumple esta propiedad. En cualquier caso, al ejecutar esta propiedad QuickCheck encuentra un error, el cual después de aplicar el proceso de reducción, es transformado a `"@ @"`:

```
> eqc:quickcheck(lstd_template_eqc:prop_string()).
Failed! After 1 tests.
[{var,"<J",[]}]

Template: "@<J@"
Substs : [{"<J",[]}]
Expected: []
Got : "@<J@"
Shrinking..(2 times)
[{var," ",[]}]

Template: "@ @"
Substs : [{" ",[]}]
Expected: []
Got : "@ @"
```

Hasta ahora, la implementación simple de la función `string` fue suficiente para cumplir la propiedad `prop_string_empty_list`. Ahora, sin embargo, hay un caso de prueba que falla, lo cual quiere decir que se necesita implementar más funcionalidad para que se ejecute de forma exitosa.

Es importante señalar que, aunque `prop_string` es la propiedad clave de la librería de plantillas, especificar antes el *parser* y el *scanner* permite al programador trabajar en la implementación en pequeños y fáciles pasos. Intentar escribir la

propiedad `prop_string` en las fases iniciales del desarrollo requeriría demasiado esfuerzo de programación antes de que las pruebas pudieran ser ejecutadas satisfactoriamente. En este caso, el desarrollo probablemente finalizaría con una solución similar a la propuesta, aunque sin tener pruebas automáticas para respaldar el funcionamiento del *scanner* y el *parser*.

Con la siguiente implementación de la función `string`:

```
string(String, Subs)->
 apply_subs(parse(tokens(String)), Subs).

apply_subs(Parsed, Subs)->
 lists:concat([to_string(X,Subs) || X<-Parsed]).

to_string({text, Text}, _Subs) ->
 Text;
to_string({var, Var}, Subs) ->
 lstd_lists:keysearch(Var, Subs).
```

la propiedad `prop_string` se cumple, pero cuando se ejecutan el resto de propiedades, QuickCheck encuentra un contraejemplo para una propiedad que anteriormente sí que se cumplía (`prop_string_empty_list`):

```
prop_string_empty_list: ...Failed! Reason:
{'EXIT',{{case_clause,{not_found,"_",[]}},
      [{eqc,'-forall/2-fun-4-',2},
       ...]}}
After 19 tests.
```

Como este mensaje de error no es muy informativo, se cambia esta propiedad para capturar errores e imprimirlos en pantalla usando la macro `WHENFAIL`:

```
prop_string_empty_list()->
 ?FORALL(S, ql_gen:string(),
    try
     S =:= lstd_template:string(S, [])
    catch
     Error:Reason->
      ?WHENFAIL(io:format("Error~p:~p~n", [Error,Reason]),
         false)
    end).
```

Ejecutando esta nueva propiedad, se obtiene el siguiente informe de error:

```
prop_string_empty_list: ...Failed! Reason:
...................Failed! After 20 tests.
"@Z"
Error throw:{not_found,"Z",[]}
Shrinking.(1 times)
"@"
false
```

Como se observa, el problema ocurre con el carácter `@`. Esto es debido a que éste es un carácter especial. Por tanto, en el momento actual, la función `string` es capaz de substituir variables por literales, pero no todas las entradas son válidas. Evidentemente, se podrían especificar casos negativos, como, por ejemplo, comprobar que la función `string` devuelve un error `not_found` cuando hay variables en la plantilla que no están en la lista de substituciones. Otra solución es simplemente arreglar el caso positivo cambiando la propiedad para que la función `string` se aplique únicamente a cadenas de caracteres que no contengan el símbolo `@`. Esta ha sido la solución por la que se opta en este caso:

```
prop_string_empty_list()->
 ?FORALL(S, valid_string(),
    try
      S =:= lstd_template:string(S, [])
    catch
      Error:Reason->
        ?WHENFAIL(io:format("Error~p:~p~n", [Error,Reason]),
           false)
 end).
```

5.5.2.4. Añadir soporte para el carácter @

Las propiedades definidas comprueban que la librería de plantillas funciona de la manera esperada cuando no se usa el carácter `@`. Sin embargo, usar el carácter `@` en lugares equivocados podría provocar fallos, pero ni siquiera se conoce qué clase de fallos se producen, o si éstos ocurren siempre, o sólo en algunas situaciones. En resumen, el comportamiento para cadenas de caracteres con caracteres `@` está totalmente indefinido.

El uso del carácter `@` ha sido prohibido en los nombres de las variables, y no es buena idea dar soporte para ello, pero sí que se debería permitir en las secciones de texto. Es obvio que con la implementación actual insertar caracteres `@` en las secciones de texto no va a funcionar como se espera, puesto que se comprobó anteriormente con un resultado no satisfactorio. Por tanto, antes de pensar en cómo implementar esta nueva característica, se deben actualizar las propiedades para considerar este caso. Así, después de actualizar los generadores y las funciones auxiliares, habrá casos de prueba que fallen que puedan guiar la implementación de

esta nueva funcionalidad. El primer paso es extender la representación elegida para las plantillas añadiendo un nuevo símbolo al generador de texto (`escaped_at`), que representará el carácter `@`:

```
text() ->
 eqc_gen:frequency(
    [{5, {text, valid_string()}}, {1, escaped_at}]).
```

Como se observa, se usa el generador `frequency` proporcionado por QuickCheck, el cual genera uno de los posibles valores en función del peso asignado a cada uno de ellos. Un ejemplo de representación de una plantilla generada con este generador es el siguiente:

```
[escaped_at, {var,"foo","FOO"},
 escaped_at, {text,"bar"}]
```

donde el resultado de la substitución es `"@FOO@bar"`, siendo los caracteres `@` parte del resultado final, no de las variables de substitución.

Después de actualizar el generador de plantillas, las propiedades escritas hasta ahora han dejado de funcionar. Por tanto, las implementaciones de las funciones `to_string`, `to_tokens`, `to_parsed`, `to_result` y `to_subs` deben ser adaptadas a la nueva representación de las plantillas. En concreto, la salida para `escaped_at` proporcionada por estas funciones auxiliares debe ser la siguiente:

- `to_string`: `"@@"`
- `to_tokens`: `[at,at]`
- `to_parsed`: `{text,"@"}`
- `to_result`: `"@"`
- `to_subs`: `[]`

La figura 5.5 muestra cómo se deben comportar estas funciones con un ejemplo de representación de plantilla. Como se observa, los caracteres `@` son escapados como `@@`. Por ejemplo, `"a@@"` se parsea como `[{text,"a"},{text,"@"}]` en vez de `[{text,"a@"}]`.

En cualquier caso, una vez que estas funciones auxiliares se actualicen, las propiedades implementadas definirán la siguiente versión de la librería de plantillas, la cual soportará el uso de caracteres `@` simplemente escapándolos como `@@`. Ejecutando las pruebas, es posible comprobar qué propiedades no se cumplen con la implementación actual:

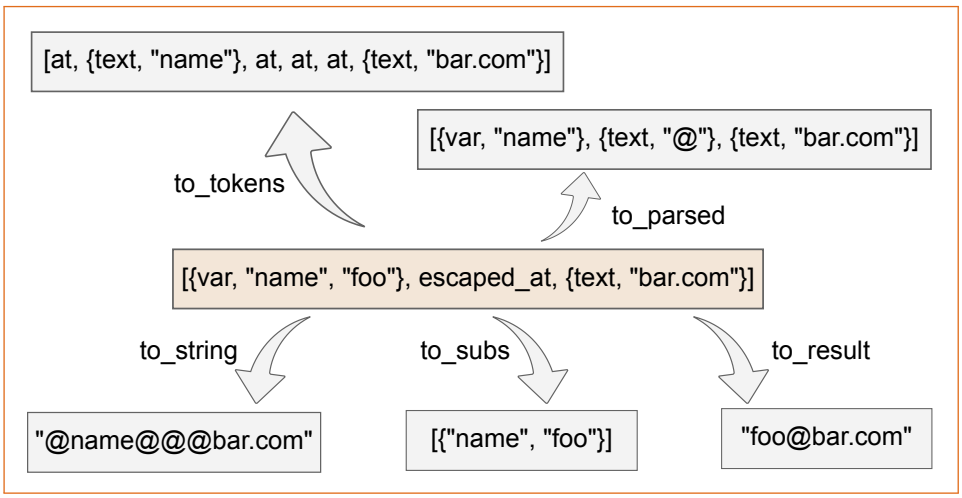


FIGURA 5.5: *Funciones* `to_string`, `to_substs`, `to_result`, `to_tokens` *y* `to_parsed` *para el carácter* `@`

```
prop_parse: ..Failed! After 12 tests.
[{text,"5"},escaped_at]
Shrinking.(1 times)
[escaped_at]
prop_string: Failed! After 1 tests.
[escaped_at]
```

Esto significa que el *scanner* todavía está funcionando, pero el *parser* ha dejado de funcionar y, por tanto, la implementación de la función `string`. En concreto, una entrada `@@` debería resultar en `@`, pero se devuelve `""`:

```
Template: "@@"
Substs : []
Expected: "@"
Got : []
```

Como las propiedades definidas hasta ahora estaban cubriendo únicamente cadenas de caracteres sin el carácter `@`, el comportamiento para cadenas de caracteres con `@` estaba todavía sin definir, por lo que la obtención de este error no debería sorprender. Ahora que este nuevo comportamiento está definido a través de una propiedad, y que hay un contraejemplo, se debe modificar la implementación del *parser* para que todos los casos de prueba generados se ejecuten de la manera esperada. A este nivel, la primera estrategia podría ser realizar una serie de comprobaciones rápidas con pequeños cambios y observar el resultado. Por ejemplo, si se cambia la función `parse` de esta forma:

```
parse([at, at | T], Terminal)->
  [{text,"@"} | parse(T, Terminal)];
parse([at|T], Terminal)->
  parse(T, switch_terminal(Terminal));
parse([{string, S} | T], Terminal)->
  [{Terminal, S} | parse(T, Terminal)];
parse([], _)->
  [].
```

la propiedad `prop_parse` determina que la implementación sigue sin ser válida:

```
> eqc:quickcheck(lstd_template_eqc:prop_parse()).
..............Failed! After 16 tests.
[{var,"\r","\""},{var,"\f(d",[]}]
Shrinking....(4 times)
[{var," ",[]},{var," ",[]}]
false
```

Esto significa que esta propiedad no se cumple para el contraejemplo `"@ @@ @"`. Para depurar este problema, es posible escribir de nuevo la propiedad usando la macro `WHENFAIL` proporcionada QuickCheck, o simplemente ver en la consola lo que está sucediendo. La primera opción es, sin duda, mejor, ya que puede ser útil en el futuro, pero para este ejemplo se optará por la segunda por simplicidad:

```
> lstd_template:parse(lstd_template:tokens("@ @@ @")).
[{var," "},{text,"@"},{var," "}]
```

El *parser* está interpretando, de forma errónea, la secuencia `@@` como un texto. El problema es que las secuencias `@@` deben ser interpretadas como texto sólo cuando no se está esperando por un carácter `@` que cierra una variable que ya ha sido abierta con otro carácter `@`:

```
parse([at, at| T], Terminal = text) ->
  [{text, "@"} | parse(T, Terminal)];
parse([at | T], Terminal) ->
  parse(T, switch_terminal(Terminal));
parse([{string, S} | T], Terminal) ->
   [{Terminal, S} | parse(T, Terminal)];
parse([], _) ->
   [].
```

Con este cambio, todas las propiedades se vuelven a cumplir y, por tanto, la implementación actual de la librería de plantillas ya soporta el uso de caracteres `@`, escapados, dentro del texto.

5.5.2.5. Análisis de los generadores usados

Aunque, aparentemente, la implementación actual de la librería de plantillas produce los resultados esperados, y cumple todas las propiedades escritas hasta ahora, todavía existe un problema oculto que algunas veces causa que la ejecución de las propiedades QuickCheck falle con el siguiente error:

```
prop_string: .....Failed! After 48 tests.
[{var,"?%",[]},
 {var,"W<ZY","y)8uFE"},
 {text,":5OH\r'"},
 {var,"sPtd4BH","` 2"},
 {var,"k","(\eg\v"},
 {var,"k","\e4<?bi"}]

Template: "@%@@<ZY@:5OH\r'@sPtd4BH@@k@@k@"
Substs : [{"%",[]},
          {"<ZY","y)8uFE"},
          {"sPtd4BH","` 2"},
          {"k","(\eg\v"},
          {"k","\e4<?bi"}]
Expected: "y)8uFE:5OH\r'` 2(\eg\v\e4<?bi"
Got : "y)8uFE:5OH\r'` 2(\eg\v(\eg\v"
Shrinking.....(5 times)
[{var,"k",[]},{var,"k"," "}]

Template: "@k@@k@"
Substs : [{"k",[]},{"k"," "}]
Expected: " "
Got : []
```

Una vez que QuickCheck encuentra este error, después de aplicar el proceso de reducción, el problema es bastante claro: la representación de las plantillas es inconsistente, ya que no es posible especificar diferentes valores de substitución para el mismo nombre de variable. Por ejemplo, si se genera la siguiente representación `[{var,"k","x"}, {var,"k","y"}]`, la cual representa la plantilla `@k@@k@`, el resultado esperado sería `xy`, pero obtiene `xx`.

Existen múltiples formas de resolver esto, por ejemplo, es posible filtrar nombres repetidos de variables para que en la lista de substituciones cada nombre de variable aparezca únicamente una sola vez. No obstante, que este problema sea detectado muy pocas veces por QuickCheck se debe a que la causa que lo origina es muy improbable que aparezca con la implementación actual de los generadores de datos utilizados en las propiedades. En concreto, debe suceder que se genere dos veces el mismo nombre de variable con diferentes valores de substitución, lo cual tiene

una baja probabilidad de suceder. Por tanto, este hecho revela un problema en la especificación, puesto que QuickCheck ha tenido problemas para encontrar el error. Esto significa que la distribución generada por el generador de plantillas debe ser modificada para hacer más probable la generación de plantillas que provoquen este tipo de fallos.

Adicionalmente, las propiedades usadas para generar los casos de prueba no están cubriendo todos los casos de la implementación. Por ejemplo, la función `to_substs` devuelve las substituciones a realizar en el mismo orden en el que aparecen en la representación de la plantilla, por lo que tampoco se está probando lo que ocurre cuando las substituciones no están en el mismo orden en el que están las variables que aparecen en la plantilla. Incluso, éstas se repiten tantas veces como aparecen en la representación. Por ejemplo, con la siguiente representación de plantilla:

```
[{var,"VAR1","double"},
 {var,"VAR2","single"},
 {var,"VAR1","double"}]
```

QuickCheck generará la siguiente expresión:

```
string("@VAR1@@VAR2@@VAR1@",
   [{"VAR1","double"},{"VAR2","single"},
    {"VAR1","double"}]) =:= "doublesingledouble"
```

lo cual tiene sentido según la implementación, pero el segundo `"VAR1"` es ignorado, por lo que no representa la prueba esperada. Una posible solución es crear un generador que baraje (y posiblemente repita de forma aleatoria), las entradas de la lista de substitución.

5.5.3. Resultados de aplicar la metodología de pruebas

Con este caso de estudio se ha ilustrado cómo el desarrollo dirigido por las pruebas y las pruebas basadas en propiedades funcionan muy bien juntos. Para ello, se ha utilizado la versión Erlang de QuickCheck como herramienta de pruebas. El uso de esta herramienta permite escribir propiedades en código Erlang usando una serie de librerías y macros proporcionadas por la propia herramienta QuickCheck, las cuales se ejecutan a través de la generación automática de cientos o miles de casos de prueba a partir de dichas propiedades. Además, se ha podido comprobar cómo el proceso de reducción de contraejemplos encontrados ayuda al programador a encontrar el defecto que ha causado un fallo, y que a su vez provocó que una determinada propiedad no se cumpla.

Aunque no se ha comentado explícitamente en la descripción de la implementación de la librería de plantillas, es importante mencionar el hecho de que Quick-

Check genera casos de prueba aleatorios a partir de una especificación de pruebas en forma de propiedades. De esta forma, al contrario que en las aproximaciones en las que se especifican los casos de prueba manualmente, en este caso es muy común que no se utilicen los mismos casos de prueba en cada ejecución de las mismas. Esto puede causar que sólo en algunas ejecuciones de las pruebas se produzcan fallos. Por esta razón, la herramienta QuickCheck ofrece funciones, como `counterexample` o `recheck`, que permiten ejecutar de nuevo casos de prueba que han causado que una propiedad no se cumpla.

Siguiendo con la comparación con las aproximaciones en las que los casos de prueba se especifican manualmente, el desarrollo dirigido por propiedades reduce el riesgo de elegir un mal conjunto de casos de prueba para conducir el desarrollo, o incluso de un buen conjunto de casos de prueba que se convierten en un conjunto malo después de un cambio en el código. Así, es más probable que un generador cubra nuevos casos límite introducidos en cambios futuros del código. Esto es debido a que las propiedades suelen ser mucho más independientes de la implementación que los ejemplos escritos manualmente por el programador. Por ejemplo, en el caso de estudio presentado, se muestra cómo los caracteres `@` llegan a ser especiales en la mitad de la implementación, y el generador empieza a encontrar un número de problemas en la implementación a causa de esto. Si se usase una aproximación manual para probar la implementación, los nuevos casos de prueba deberían haber sido escritos una vez que este comportamiento haya sido añadido.

Por otra parte, los programadores deben ser cuidadosos para conseguir una buena distribución de los datos generados por los generadores implementados. Por ejemplo, en el caso de estudio mostrado, un defecto en la especificación quedó oculto durante todo el proceso de implementación porque la causa que lo originaba era extremadamente improbable que ocurriese con el generador de plantillas inicial. En el desarrollo dirigido por las pruebas este problema no existe, puesto que no existen generadores de datos, pero al dirigir el desarrollo por propiedades es crítico validar la distribución de datos usada, por ejemplo, usando funciones ofrecidas por la propia herramienta QuickCheck, como `collect` o `aggregate`, que permiten mostrar la distribución de valores generados durante la realización de las pruebas.

Por último, el desarrollo dirigido por propiedades ayuda a una mejor selección de propiedades escritas para especificar el código, en contraposición a escribir las propiedades después de la implementación. Esto es debido a que, en un desarrollo dirigido por propiedades, el programador está seguro de que la propiedad prueba algo que está fallando y que debería funcionar. En otras palabras, es menos probable escribir propiedades triviales que siempre se cumplen y que dan a los programadores una falsa sensación de seguridad sobre el código que supuestamente se está probando.

5.6. Resumen

La realización de las pruebas de unidad por parte del programador es una tarea fundamental en el desarrollo de software. Sin embargo, aunque toda la comunidad informática está de acuerdo con esta afirmación, muchas veces éstas no son realizadas con todo el rigor necesario o son omitidas totalmente, la mayoría de las veces argumentando que no queda tiempo para realizarlas. Cuando esto ocurre, se corre el riesgo de que aparezcan defectos en el software en fases posteriores del desarrollo, o incluso cuando éste esté siendo usado por el cliente, con los costes asociados que esto conlleva.

Dentro de las diferentes metodologías de desarrollo, el desarrollo dirigido por las pruebas evita el problema de que las pruebas no se implementen, puesto que éstas deben escribirse incluso antes que la propia implementación. Esta metodología es idónea para la realización de las pruebas de unidad. Sin embargo, su uso se basa en escribir las pruebas como ejemplos concretos, lo cual puede provocar que se olviden casos de prueba límite en algunas situaciones. En estos casos, usar una aproximación basada en propiedades permite generalizar los casos de prueba a través de propiedades que describen el sistema a probar.

De esta forma, la metodología propuesta parte de una combinación entre el desarrollo dirigido por las pruebas y las pruebas basadas en propiedades. Esta nueva metodología, a la que se ha llamado *desarrollo dirigido por propiedades*, puede considerarse como la evolución natural del desarrollo dirigido por las pruebas cuando se usa una aproximación basada en propiedades. La idea del desarrollo dirigido por propiedades consiste en que antes de escribir la implementación para un nuevo requisito, es necesario especificar dicho requisito, escribiendo propiedades que el código debe cumplir para, de esta forma, poder encontrar un contraejemplo que no cumpla las propiedades especificadas y, finalmente, escribir el código necesario para arreglar todos los contraejemplos encontrados.

6

APIs de integración

6.1. Introducción

Los sistemas software complejos suelen estar formados por componentes que se integran entre sí para implementar la funcionalidad requerida. La integración de dichos componentes se suele realizar a través de APIs de integración proporcionadas por cada uno de ellos. Una API de integración es una colección de operaciones que permiten a un componente externo acceder a una serie de funcionalidades que proporciona otro componente (el que ofrece dicha API). De esta forma, las APIs de integración facilitan el intercambio de mensajes o datos entre componentes software.

Por otro lado, los protocolos de comunicación describen las secuencias válidas de operaciones de una o varias APIs de integración que se deben invocar para llevar a cabo alguna tarea de alto nivel. Así, mientras que una API de integración define una serie de operaciones, los tipos de datos de entrada y salida, las condiciones en las que cada operación puede ser invocada (precondiciones), y los efectos que conlleva ejecutar cada operación (postcondiciones); los protocolos de comunicación, por su parte, definen cómo se deben usar las APIs de integración, por ejemplo, cuál debe ser el orden en el que se deben llamar a las operaciones para realizar alguna tarea de alto nivel.

Las pruebas de las implementaciones de este tipo de APIs son una parte importante dentro del desarrollo de sistemas complejos. Implementaciones incorrectas de APIs de integración producirán como resultado integraciones incorrectas entre los diferentes componentes de un sistema software y, por tanto, provocarán que el sis-

tema resultante no funcione de la manera esperada. Además, en muchas ocasiones, las APIs de integración son las interfaces que ofrece un producto a programadores externos, quienes desarrollan software propio usando el componente que ofrece dicha API de integración. Si estas interfaces fallan, además de afectar al software creado por estos programadores externos, es probable que éstos también elijan otras alternativas, produciendo así una pérdida de clientes. De esta forma, las pruebas de APIs de integración suponen aumentar la estabilidad de los productos software, y además de mejorarlo internamente (producto software más fácil de desarrollar, probar y mantener), lo mejoran de forma externa, aumentando su éxito en el mercado.

Probar la implementación de una API de integración consiste en comprobar el funcionamiento de las diferentes operaciones que forman parte de dicha API, así como de las secuencias de llamadas a dichas operaciones, normalmente desde una perspectiva de caja negra. Desde este punto de vista, aunque los conceptos explicados en el capítulo 5 son totalmente válidos para probar implementaciones de APIs de integración, en este caso se dan una serie de peculiaridades que provocan que sea posible usar técnicas más específicas que se adaptan mejor a la realización de las pruebas de este tipo de sistemas. Es por ello que, en este capítulo, se expone una metodología para probar implementaciones de APIs de integración. Puesto que las APIs de integración evolucionan rápidamente, y es común disponer de varias implementaciones de una misma API de integración, la aproximación propuesta se basa en utilizar la especificación de la API de integración en vez de la propia implementación de la misma para, a través de una aproximación basada en propiedades, y basándose en la arquitectura propuesta por el estándar TTCN-3 [197, 367], generar automáticamente casos de prueba que permitan probar las implementaciones concretas de dicha API de integración.

Los contenidos de este capítulo se estructuran de la siguiente manera. La sección 6.2 explica en qué consisten las pruebas de APIs de integración de una manera genérica, describiendo algunas tecnologías y aproximaciones usadas para este fin, y que sirven de inspiración para la aproximación de pruebas desarrollada, como son el estándar TTCN-3 (sección 6.2.1) y las pruebas basadas en modelos (sección 6.2.2). Posteriormente, en la sección 6.3, se describen las máquinas de estados QuickCheck, que constituyen la base de la técnica de pruebas desarrollada en este capítulo para probar APIs de integración. A continuación, en la sección 6.4, se explica dicha técnica, la cual está basada en propiedades, como parte de una metodología de pruebas, para lo que se utiliza un caso de estudio que ilustra su aplicación. Una vez explicados los detalles de esta metodología, se exponen, en la sección 6.5 (y usando de nuevo un caso de estudio), una variante de la aproximación desarrollada en la que se utilizan lenguajes de modelado, en concreto, UML y OCL, para especificar el comportamiento de la API de integración, en vez de escribir las propiedades manualmente. Finalmente, en la sección 6.6, se realiza un resumen de los contenidos de este capítulo.

6.2. Las pruebas de APIs de integración

Probar una implementación de una API de integración tiene como objetivo comprobar que cada una de las operaciones que forman parte de dicha API se comportan de la forma esperada. Así, para realizar esta tarea es necesario invocar cada una de las operaciones, con diferentes combinaciones para los parámetros de entrada, analizando el comportamiento tanto para valores "normales" (esperados) de dichos parámetros, como para valores inválidos y valores frontera.

Para APIs de integración proporcionadas por componentes sin estado, como puede ser, por ejemplo, una librería matemática, realizar llamadas aisladas a cada una de las operaciones puede ser suficiente. Sin embargo, muchas APIs de integración son ofrecidas por componentes con estado, en los que el resultado de invocar una operación puede variar en función de otras operaciones invocadas anteriormente, por lo que no basta con invocar cada una de las operaciones de manera independiente, sino que es necesario realizar combinaciones de secuencias de llamadas a las diferentes operaciones para comprobar que las tareas de alto nivel pueden ser llevadas a cabo.

Existen diferentes aproximaciones que pueden ser utilizadas para generar las combinaciones de secuencias de operaciones a ejecutar. Por ejemplo, es posible analizar manualmente las dependencias entre las diferentes operaciones y crear secuencias de operaciones teniendo en cuenta dichas dependencias. O, incluso, es posible capturar secuencias de llamadas a operaciones realizadas por aplicaciones reales en uso para añadirlas a las pruebas. Sin embargo, los casos de prueba obtenidos con estas aproximaciones, aunque pueden ayudar a proporcionar una cierta confianza sobre la implementación de la API de integración, se basan en probar escenarios comunes. Por tanto, otras combinaciones de operaciones, aún siendo posible que sean realizadas por las aplicaciones, no serán ejecutadas ni, por tanto, probadas en absoluto. Es por ello que otras aproximaciones de pruebas consisten en generar, con soporte de herramientas, secuencias aleatorias con todas las operaciones disponibles, teniendo en cuenta una serie de precondiciones (y postcondiciones) para cada operación.

Una de las principales diferencias entre las pruebas de unidad (descritas en el capítulo 5) y las pruebas de APIs de integración es que, mientras que las primeras las realizan los mismos programadores que implementan el código fuente, las pruebas de APIs de integración pueden ser llevadas a cabo por equipos de probadores especializados en la realización de este tipo de pruebas. De esta forma, mientras que los programadores tienen acceso al código fuente, escrito por ellos mismos, los probadores de software, en este caso, normalmente no tienen acceso a dicho código, y tratan al sistema como una caja negra, usando únicamente su especificación. Un equipo independiente de probadores permite, además, evitar el efecto de la posible "parcialidad" cuando una persona prueba su propio código fuente [365].

Las *pruebas de caja negra*, en contraposición a las *pruebas de caja blanca*, tratan al sistema a probar desde un punto de vista externo, sin realizar ningún tipo de suposición sobre su estructura interna. De esta forma, las pruebas de caja negra suelen evaluar el comportamiento del sistema a probar comparando las salidas obtenidas para una serie de entradas, con las salidas esperadas para esas mismas entradas. Además, puesto que una API de integración representa la forma de acceder a un componente desde otros componentes, las pruebas de caja negra suelen ser la aproximación adecuada para probar este tipo de implementaciones.

Por otro lado, mientras que en las pruebas de unidad se prueban unidades de software independientes, normalmente aislando las funcionalidades específicas de los módulos a probar; las pruebas de implementaciones de APIs de integración suelen requerir dependencias del contexto o la propia configuración del entorno en el que el componente va a funcionar. De esta forma, en algunas ocasiones, es necesario que el componente que implementa la API de integración a probar se instale en uno o varios entornos de pruebas que tengan en consideración los escenarios en los que la API de integración puede ser usada.

Con respecto a las herramientas usadas en este tipo de pruebas, éstas suelen ser las mismas que las usadas en las pruebas de unidad, como, por ejemplo, las herramientas xUnit. Además, en este caso, es más común usar extensiones de estas herramientas cuando se prueban implementaciones de APIs de integración, como es el caso de XMLUnit [77] o HttpUnit [31], útiles para probar APIs de integración implementadas como servicios web. También existen lenguajes específicos, como es el caso de TTCN-3 [197, 367], que permiten a los probadores de software escribir especificaciones de pruebas sin tener en cuenta la estructura interna del sistema a probar.

6.2.1. El estándar para control y ejecución de pruebas TTCN-3

TTCN-3 [197, 367] es un lenguaje estandarizado, desarrollado y mantenido por la ETSI [25], oficialmente presentado el año 2000. Está basado en anteriores versiones de TTCN, en concreto, TTCN-2 (presentado en los años 80 y, a su vez, sucesor de TTCN-1), y está diseñado específicamente para la realización de pruebas, orientado sobre todo a pruebas de caja negra. TTCN-3 es, así, un lenguaje de especificación de pruebas, pensado y diseñado específicamente para este fin. Por tanto, a diferencia de los lenguajes de programación de propósito general, TTCN-3 permite el uso de estructuras y componentes orientados a la realización de pruebas que hacen que el código de pruebas sea compacto y fácil de entender [332].

El lenguaje TTCN-3 ha sido creado para probar protocolos de telecomunicación (3G, Bluetooth, IPv6, WiMAX, etc.) y, de esta forma, al igual que su predecesor TTCN-2, el cual fue desarrollado por ISO como parte de una metodología de pruebas para la arquitectura OSI [157], la industria de las telecomunicaciones es el ám-

bito donde más se usa, tanto en las pruebas de sistema como en las pruebas de aceptación. Así, ha sido usado en las pruebas de protocolos de IEEE e IETF.

Existen otros ámbitos, como el de la automoción (por ejemplo, dentro del consorcio AUTOSAR [199, 362]), aviónica [333], médico (por ejemplo, para los protocolos HL7 eHealth [171, 351]) o, incluso, relacionado con ferrocarriles [119, 251], donde TTCN-3 ha sido o está siendo usado para probar diferentes sistemas software. Además del ámbito industrial, TTCN-3 también ha sido usado en el ámbito académico, donde se han publicado trabajos de investigación que proponen el uso de TTCN-3 para pruebas de integración de componentes [296], pruebas de aplicaciones web [303, 334], servicios web [318, 371] e incluso definiendo extensiones para usar este lenguaje en otros tipos de pruebas, como son las pruebas de rendimiento [319].

El lenguaje TTCN-3 permite codificar la especificación de pruebas en diferentes formatos de presentación. Por una parte, el formato de presentación *textual* es el formato base, conocido como la *notación core* [178] de TTCN-3. Muchas construcciones de este lenguaje son similares a otros lenguajes de programación de propósito general (tipos de datos, variables, constantes, funciones, instrucciones de control de flujo, etc.). Sin embargo, TTCN-3 aporta nuevos conceptos orientados a la realización de pruebas no existentes en muchos otros lenguajes de programación. La comparación de valores mediante el uso de patrones de coincidencia de datos personalizados, una arquitectura distribuida de pruebas, o la ejecución concurrente de componentes de prueba con soporte para definir veredictos de prueba son algunas de las funcionalidades que aporta TTCN-3 que no son comunes en otros lenguajes de programación de propósito general.

Por otra parte, es posible usar otros formatos de presentación, los cuales siempre pueden ser transformados al formato textual preservando la semántica. Aunque es posible crear formatos de presentación específicos para aplicaciones específicas o incluso formatos propietarios, existen dos formatos que han sido estandarizados: el formato tabular [175], donde los casos de prueba se especifican rellenando una serie de tablas; y el formato gráfico [176], el cual usa diagramas, muy similares a los diagramas de secuencia UML, para especificar los casos de prueba.

Una de las características principales del lenguaje TTCN-3 es que permite un alto nivel de abstracción a la hora de realizar las pruebas, puesto que es independiente del lenguaje de programación que ha sido usado para implementar el sistema a probar. De este modo, las pruebas siempre se codifican usando el lenguaje TTCN-3, sin importar la naturaleza del sistema a probar, conformando así un *conjunto de pruebas abstractas* (ATS). Escribir las pruebas del software usando el lenguaje TTCN-3 implica, por tanto, escribir un ATS en uno de los formatos de presentación soportados, siendo el formato textual el más habitual. Este conjunto de pruebas abstractas debe ser procesado, o bien por un compilador TTCN-3 que traduzca el ATS a código C, Java o algún otro tipo de lenguaje, enlazado con librerías de ejecución propietarias

de TTCN-3; o bien puede usarse un intérprete TTCN-3 que genere algún tipo de *bytecode* que pueda ser interpretado por una máquina virtual TTCN-3.

De esta forma, el uso de TTCN-3 conlleva la necesidad de disponer, como mínimo, de un compilador o intérprete TTCN-3 que permita obtener un conjunto de pruebas ejecutables a partir de la especificación de pruebas; así como de un entorno de ejecución de pruebas. Actualmente, existen una serie de herramientas, muchas de ellas comerciales, como OpenTTCN Tester [52], TTCN-3 toolbox [18] o Elvior [22], y algunas no comerciales, como LoongTesting [70] o Broadbit [14], que soportan totalmente o parcialmente la especificación de TTCN-3 para obtener un conjunto de pruebas ejecutables. Además, también existen herramientas que facilitan el uso de TTCN-3. Por ejemplo, hay entornos integrados de desarrollo (IDE) que facilitan tanto la implementación de pruebas y adaptadores (los cuales, como se comentará posteriormente son esenciales en la arquitectura necesaria para realizar las pruebas con TTCN-3), la ejecución de las propias pruebas con TTCN-3, e incluso la inspección de los resultados obtenidos en las mismas [52, 68, 70].

Cabe mencionar que a la hora de usar el lenguaje TTCN-3, siempre es necesario el uso de un *adaptador* que se encargue de comunicar el conjunto de pruebas abstractas con el sistema a probar. Los adaptadores se suelen escribir manualmente en algún lenguaje de programación de propósito general, como Java, C, C++, o C#, aunque también se han creado herramientas que permiten simplificar la generación de los mismos [62]. En cualquier caso, los adaptadores que se deben implementar cuando se usa TTCN-3 deben seguir una estructura específica. En concreto, éstos deben tener, al menos, las dos siguientes capas diferenciadas (figura 6.1):

- Una capa de *codificación*, que codifica los datos procedentes del ATS en un formato entendible por el sistema a probar; y decodifica los mensajes que provienen de dicho sistema a probar al formato esperado en el ATS.
- Una capa de *adaptación de sistema*, que se encarga de comunicarse directamente con el propio sistema a probar, enviando los mensajes codificados por la capa de codificación, y recibiendo los mensajes que envía el sistema a probar para ser decodificados por la misma capa de codificación.

El resultado de usar TTCN-3 es la obtención de un *conjunto de pruebas ejecutables* (ETS), que necesita usar un adaptador para poder ser ejecutado contra el sistema a probar. Este conjunto de pruebas ejecutables se ejecutan a través del sistema de ejecución de TTCN-3 (TE), el cual puede ser una librería de ejecución propietaria de TTCN-3 en caso de usar un compilador de código TTCN-3, o una máquina virtual que use *bytecode* TTCN-3 en caso de usar un intérprete TTCN-3.

Por tanto, el uso de TTCN-3 como lenguaje de especificación de pruebas permite escribir pruebas de más alto nivel, abstrayendo los detalles de implementación del sistema a probar. De esta forma, los casos de prueba se basan en la especificación del sistema a probar en lugar de en la implementación del mismo. Una de

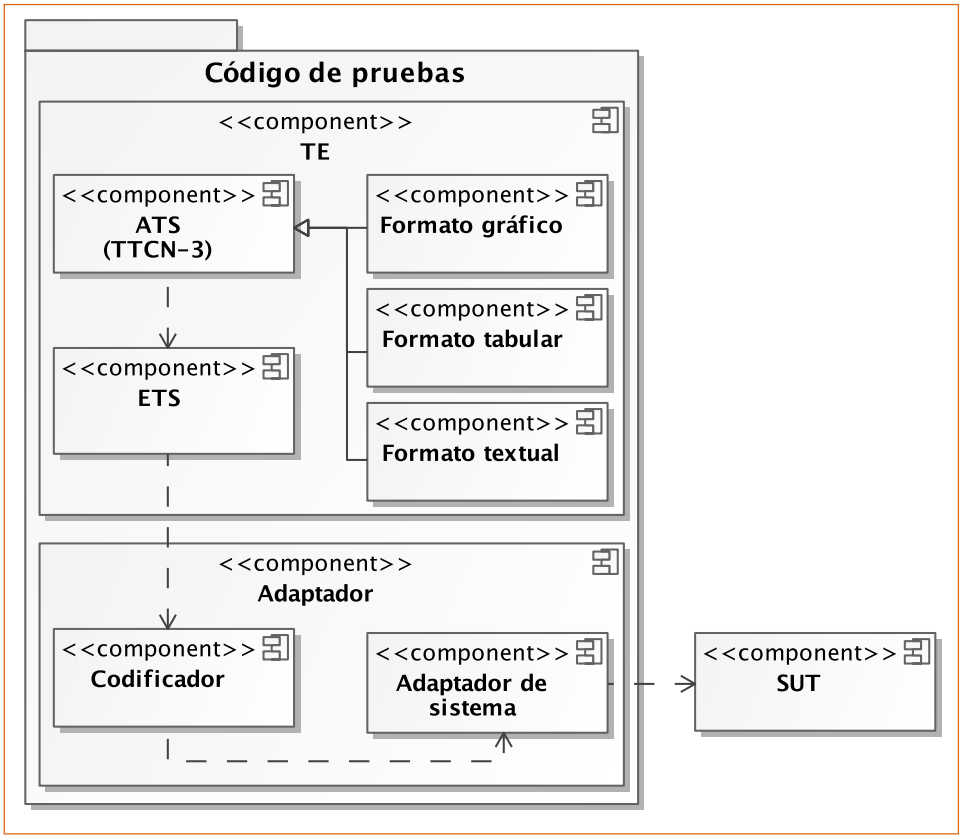


Figura 6.1: *Arquitectura para realizar las pruebas usando el estándar TTCN-3*

las ventajas de usar esta aproximación es que las especificaciones de pruebas son menos volátiles que aquellas basadas en la implementación del sistema a probar, puesto que la especificación de un sistema suele cambiar menos que su implementación. Así, en el caso de las APIs de integración, si la especificación de la API no cambia, el conjunto de pruebas abstractas tampoco cambiará aunque la implementación se modifique. Por ejemplo, cambiar el tipo de dato de respuesta, como puede ser dejar de devolver ficheros XML para devolver ficheros JSON en una API de integración implementada como un servicio web, no implicaría un cambio en el conjunto de pruebas abstractas siempre y cuando la especificación de las operaciones sea la misma. Por otra parte, si la especificación de la API de integración cambia, además de cambiar el conjunto de pruebas abstractas, es posible que haya que cambiar el adaptador manualmente. En este último caso, aunque el conjunto de pruebas ejecutables también será diferente, éstas se generarán automáticamente.

Por último, es importante mencionar que aunque la arquitectura a seguir cuando se usa TTCN-3 es más compleja que la necesaria al usar herramientas xUnit, ésta favorece una separación de roles útil cuando el sistema a probar es una implementación de una API de integración. Así, mientras que las pruebas de unidad son siempre realizadas por los programadores que codifican el sistema, las pruebas

de las APIs de integración podrían ser realizadas por un equipo de probadores de software independientes que pueden no conocer los detalles de implementación del sistema a probar. Si se usa TTCN-3, este equipo podría especializarse en el uso de únicamente este lenguaje, independientemente de cómo se haya implementado el sistema a probar. Este nivel de abstracción, ofrecido por la arquitectura necesaria al usar el lenguaje TTCN-3, es uno de los motivos por los que la técnica propuesta se basa en esta arquitectura.

6.2.2. Las pruebas basadas en modelos

Otra de las aproximaciones usadas en las pruebas de APIs de integración, y que se usará como base en la aproximación desarrollada, son las *pruebas basadas en modelos* [164, 348, 349]. Las técnicas de pruebas basadas en modelos automatizan el diseño de pruebas de caja negra a partir de *modelos* que describen aspectos, normalmente funcionales, del sistema a probar. Este proceso incluye tanto la generación de datos de entrada para los casos de prueba, la generación de secuencias de llamadas a las operaciones del sistema a probar, y la comprobación de que dichas operaciones se han ejecutado de la forma esperada. En definitiva, las pruebas basadas en modelos abarcan todo el proceso necesario para la generación automática de casos de prueba ejecutables a partir de un modelo del sistema a probar.

Basándose en los 5 niveles de automatización descritos en [255] para clasificar las técnicas de pruebas, las pruebas basadas en modelos están en el nivel 1, que representa aquellas técnicas de pruebas totalmente automatizadas. Así, en este caso, no se escriben directamente los casos de prueba, sino que se crean modelos que describen el comportamiento esperado del sistema a probar, los cuales capturan los requisitos del mismo, para generar automáticamente dichos casos de prueba.

En concreto, el proceso de generación de pruebas en una aproximación basada en modelos se realiza siguiendo los siguientes pasos descritos en la figura 6.2:

1. **Modelar el sistema a probar:** el primer paso en este tipo de técnicas es la creación de un modelo abstracto del sistema a probar. Los modelos usados en este tipo de aproximaciones son abstractos puesto que son descripciones esquemáticas del sistema que se obtienen a partir de los requisitos a probar. De esta forma, los modelos deben ser concisos y pequeños en relación al tamaño del sistema para que sean fáciles de producir, centrarse en los aspectos clave a probar, y omitir los detalles del sistema no interesantes en las pruebas. Sin embargo, por otro lado, deben ser los suficientemente detallados para que describan de forma precisa las características a probar.

2. **Generar un conjunto de pruebas abstractas a partir del modelo:** usando un criterio de selección de casos de prueba, las aproximaciones basadas en modelos generan automáticamente casos de prueba a partir del modelo inicial. Los criterios de selección de casos de prueba están relacionados con el

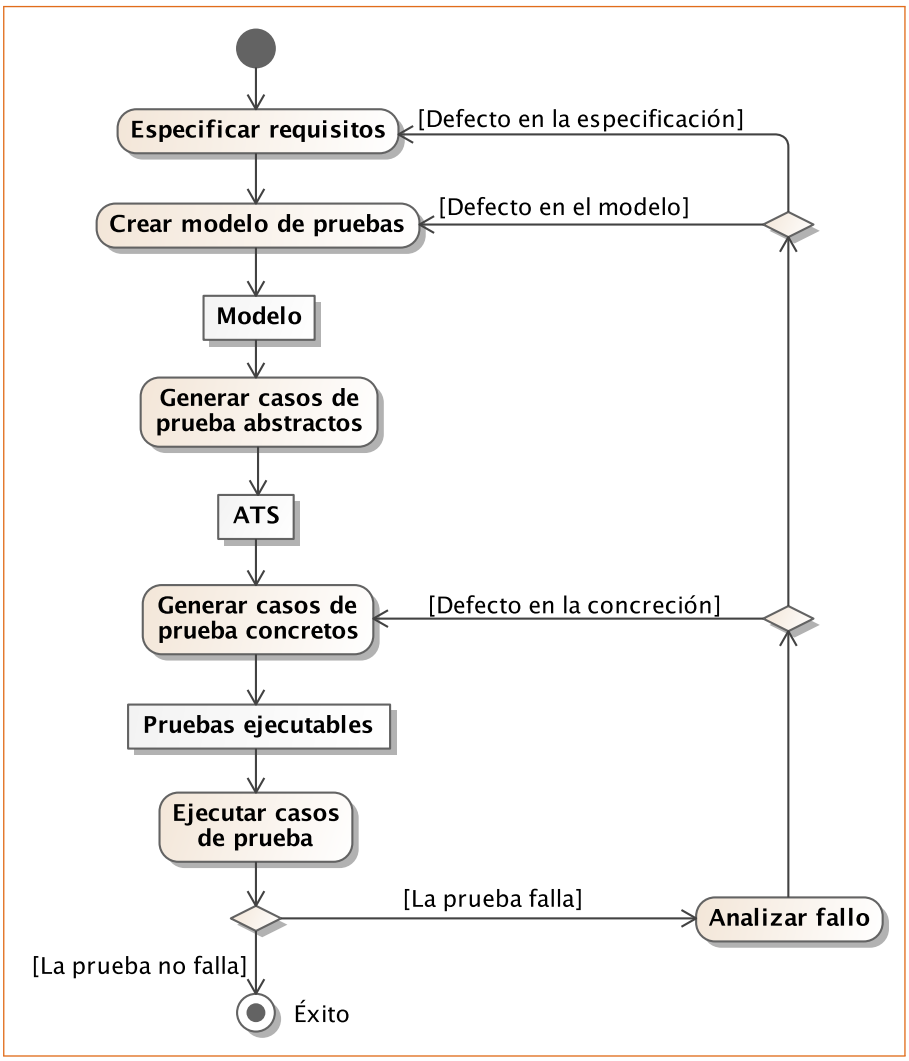


FIGURA 6.2: *Proceso para realizar las pruebas basadas en modelos*

modelo o los requisitos, nunca con el código fuente del sistema a probar, y suelen basarse en diferentes técnicas de cobertura del modelo. Así, como el conjunto de casos de prueba que se pueden generar a partir de un modelo suele ser infinito, se suelen generar un conjunto finito que cumpla un determinado criterio de cobertura. Controlar la cobertura estructural del modelo, la cobertura de los datos de entrada generados o la cobertura de los requisitos a probar son algunas de las técnicas usadas para seleccionar los casos de prueba.

Por otro lado, los casos de prueba generados a partir del modelo conforman un *conjunto de pruebas abstractas* (ATS), puesto que no contienen información concreta acerca de cómo se deben ejecutar para probar la implementación del sistema concreto, es decir, no dependen del código fuente del sistema a

probar, sino que únicamente contienen conceptos y valores del modelo. Cada uno de los casos de prueba es una secuencia de operaciones con sus valores de entrada asociados y la salida esperada.

Aunque al igual que las pruebas basadas en propiedades, las pruebas basadas en modelos generan casos de prueba a partir de una especificación del sistema a probar, éstos se generan de manera diferente. Por un lado, en las pruebas basadas en propiedades, los casos de prueba suelen generarse de forma aleatoria, mientras que en las pruebas basadas en modelos suele usarse algún tipo de criterio de cobertura. Además, los casos de prueba en las pruebas basadas en modelos son abstractos, por lo que necesitan ser procesados para poder ser ejecutados. Esto no suele suceder en las pruebas basadas en propiedades.

3. **Generar un conjunto de casos de prueba concretos que puedan ser ejecutados:** las llamadas a las operaciones reales del sistema pueden ser bastante diferentes a las operaciones abstractas que forman parte del conjunto pruebas abstractas ATS. Por tanto, es necesario preparar los casos de prueba para que puedan ser ejecutados, obteniendo así un conjunto de casos de prueba ejecutables. Para ello, es posible seguir diferentes aproximaciones, entre las que destacan las siguientes:

   - Aproximaciones basadas en adaptadores, las cuales consisten en escribir manualmente un adaptador que use el ATS para comunicarse con el sistema a probar. Así, este adaptador actúa como una especie de intérprete de los casos de prueba abstractos, realizando al menos las dos siguientes tareas: *concreción*, es decir, traducir las llamadas abstractas al sistema con sus argumentos abstractos a llamadas concretas al sistema con los valores de entrada apropiados; y *abstracción*, o lo que es lo mismo, obtener los resultados del sistema obtenidos de realizar las llamadas concretas y traducirlo a valores abstractos entendibles por el ATS.

   - Aproximaciones basadas en transformar los casos de prueba abstractos en casos de prueba concretos, por ejemplo, a través de *scripts* específicos que realicen esta función. Con esta aproximación se generara código escrito en algún lenguaje de programación estándar (Java, C, Erlang, etc.), algún lenguaje de *scripting* o cualquier otra notación que pueda producir un programa ejecutable. De la misma forma que las aproximaciones basadas en adaptadores pueden compararse con intérpretes, las aproximaciones basadas en transformaciones se asimilan a compiladores que compilan los casos de prueba abstractos en algún lenguaje de más bajo nivel.

   - Aproximaciones mixtas, que combinan las dos propuestas anteriores. Por ejemplo, es posible transformar los casos de prueba abstractos a casos de prueba concretos que sigan una estructura genérica y válida para

diferentes proyectos usando algún tipo de *script* para ello, y a su vez implementar un adaptador específico para el sistema concreto a probar que permita adaptar dichos casos de prueba concretos a las especificaciones de dicho sistema. De esta forma, se simplifican tanto el proceso de transformación como el de adaptación.

4. **Ejecutar los casos de prueba contra el sistema a probar:** tanto esta fase como la siguiente son comunes a todas las aproximaciones de pruebas, no sólo en las aproximaciones basadas en modelos. En esta fase, los casos de prueba concretos generados deben ser ejecutados contra la implementación concreta del sistema a probar. Para ello, existen dos aproximaciones básicas que se pueden seguir:

   - Ejecución de pruebas *online*: en este tipo de aproximaciones los casos de prueba se ejecutan a medida que se van generando. En este caso, el uso de adaptadores para obtener los casos de prueba concretos es lo más adecuado.

   - Ejecución de pruebas *offline*: se generan a priori una serie de casos de prueba que se ejecutarán posteriormente. Aunque las aproximaciones basadas en transformaciones son más adecuadas en este caso, también es posible usar aproximaciones basadas en adaptadores o mixtas.

5. **Analizar los resultados de prueba:** como en cualquier otro tipo de técnica de pruebas, el análisis de resultados se debe producir siempre después de la ejecución de las mismas. En las aproximaciones de pruebas tradicionales, detectar un fallo con un caso de prueba normalmente implica la existencia de un defecto en el sistema a probar, o bien en el caso de prueba. En las aproximaciones basadas en modelos, el problema podría estar presente en el sistema a probar, pero también en el modelo o en el propio adaptador. Por lo tanto, el primer paso a la hora de detectar un problema con la ejecución de las pruebas es analizar en qué parte está el problema para poder actuar en consecuencia.

Llevar a cabo estos pasos para probar un sistema utilizando una aproximación basada en modelos requiere una arquitectura como la mostrada en la figura 6.3. Como se puede observar, la parte fundamental y el pilar en este tipo de aproximaciones es el modelo del sistema a probar que se usa para generar los casos de prueba. Existen multitud de notaciones para escribir los modelos, entre las que destacan las *notaciones basadas en estados*, en las que se definen precondiciones y postcondiciones para cada operación a probar (B, UML [126, 217, 315] y OCL [129, 363], JML, Spec#, Z), y las *notaciones basadas en transiciones*, que se centran en definir transiciones entre los diferentes estados del sistema a probar (máquinas de estados finitos, diagramas de estado, sistemas de transiciones etiquetadas, autómatas, etc.).

De entre todos los formalismos que pueden ser usados en ambas aproximacio-

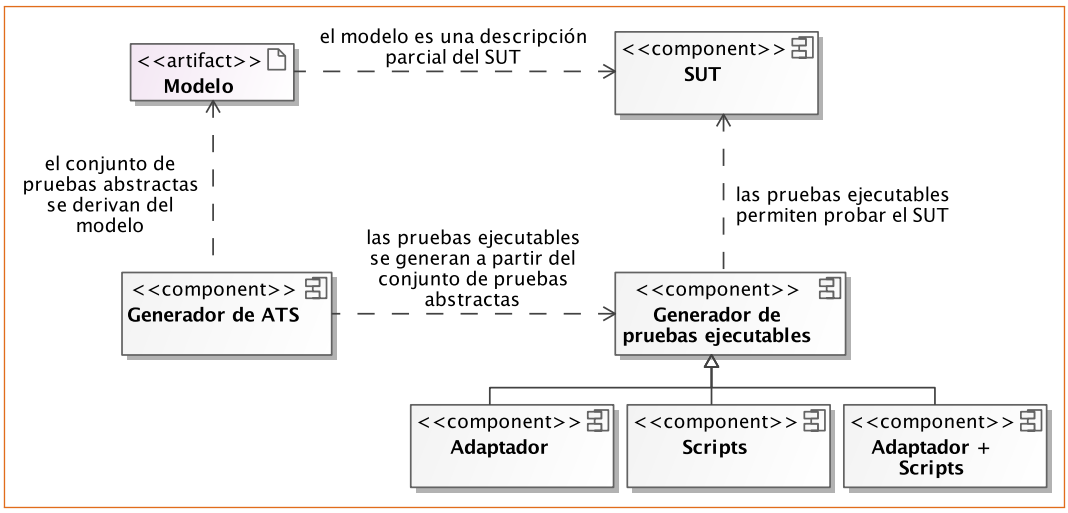


FIGURA 6.3: *Arquitectura para realizar las pruebas basadas en modelos*

nes [165, 323], destaca el uso del lenguaje de modelado UML [370]. Este lenguaje normalmente se combina con OCL para especificar precondiciones y postcondiciones. De esta forma, este tipo de restricciones OCL puede ser utilizado para generar código ejecutable [128, 140, 267] que compruebe si el sistema a probar satisface dichas condiciones, usando datos de prueba generados automáticamente [81, 114].

Por último, cabe destacar que el grado de automatización que permite este tipo de técnica de pruebas es uno de los motivos principales por los que haya sido elegida como base para complementar las pruebas basadas en propiedades en la realización de las pruebas de APIs de integración.

6.3. Las pruebas basadas en propiedades

Las APIs de integración están formadas por una serie de operaciones que, en muchas ocasiones, están relacionadas entre sí, de tal forma que la ejecución de cada una de ellas suele producir efectos colaterales en la ejecución de otras operaciones. Aunque existen ocasiones donde las operaciones de una API de integración son independientes, como son las librerías matemáticas, en general esto no es así. Dicho de otro modo, son mucho más habituales las APIs de integración ofrecidas por componentes con estado.

En estos casos, usar una aproximación basada en propiedades requiere algo más complejo que la definición de propiedades independientes para cada una de las operaciones, como se mostró en el capítulo 5 para el caso de las pruebas unitarias. Para ello, es más adecuado utilizar modelos que describan el comportamiento del sistema a probar. Con respecto a estos modelos, lo más habitual en una aproximación basada en propiedades es utilizar máquinas de estados. El estado que manejan estas

máquinas de estados suele estar formado por datos arbitrarios, por lo que el número de posibles estados suelen ser infinito. Las máquinas de estados finitos también pueden utilizarse cuando se conocen exactamente los estados del componente a probar y éstos son un número finito. En este último caso, éstas son un caso particular de las primeras. Por otro lado, las transiciones en las máquinas de estados (tanto finitos como con un número indeterminado de estados), las cuales están representadas por la ejecución de cada una de las operaciones a probar, suelen tener precondiciones y postcondiciones asociadas.

La posibilidad de utilizar máquinas de estados con un número no finito de éstos es un punto diferenciador con respecto a las pruebas basadas en modelos, donde se usan modelos abstractos del sistema a probar. En estas aproximaciones, los casos de prueba se generan algorítmicamente a partir del modelo, con el objetivo de cubrir el espacio de estados de alguna forma u otra. Este hecho implica que los modelos usados en las pruebas basadas en modelos deben ser modelos de estados finitos, o, al menos, de estados finitos con respecto a alguna relación de equivalencia.

Probar un sistema usando una máquina de estados con una aproximación basada en propiedades implica generar secuencias de transiciones a través de dicha máquina de estados, y comprobar si el sistema a probar es capaz de ejecutar cada una de ellas. Para ello, se comprueban las precondiciones y postcondiciones asociadas a las mismas. Estas últimas usualmente comprueban si el resultado obtenido al realizar una transición es el resultado esperado.

6.3.1. QuickCheck

La herramienta QuickCheck soporta el uso de modelos, en concreto, máquinas de estados (tanto finitos como con un número no determinado de estos), para definir el comportamiento del sistema a probar cuando las operaciones a probar tienen efectos colaterales en su ejecución. El uso de una máquina de estados QuickCheck conlleva la definición de un estado, inicializado al comienzo de la ejecución de las pruebas, que se irá modificando como resultado de las transiciones, las cuales están representadas por las operaciones a probar, y que tienen asociadas una serie de precondiciones y postcondiciones. De esta forma, usar una máquina de estados QuickCheck implica definir los elementos que se describen a continuación.

6.3.1.1. Definición del estado

El estado de una máquina de estados QuickCheck almacena datos que pueden ser usados en la ejecución de las pruebas: listas de identificadores creados, conjuntos de objetos, o indicadores *booleanos*, son algunos ejemplos de datos que normalmente son almacenados en dicho estado.

QuickCheck no impone ningún tipo de restricción sobre lo que se debe o puede almacenar en el estado, puesto que éste es totalmente dependiente de las pruebas

concretas que se están realizando. Normalmente, la estructura del estado suele ser un registro Erlang en el que cada uno de sus campos contiene cada uno de los valores a almacenar. Por ejemplo:

```
-record(state, { f1, f2, ...}).
```

donde los campos `f1` y `f2` contienen información a almacenar en el estado.

6.3.1.2. Inicialización del estado

Al comenzar cada prueba, QuickCheck ejecuta la función `initial_state`, la cual debe devolver los valores concretos de los datos requeridos para el estado al inicio de la ejecución de la prueba. Por ejemplo:

```
initial_state()->
 #state {
  f1 = initial_values1(),
  f2 = initial_values2(),
  ⋮
 }.
```

donde `initial_values1` e `initial_values2` representan funciones auxiliares que devolverán datos apropiados como valores iniciales para los campos `f1` y `f2`, respectivamente. Obviamente, no es necesario usar siempre funciones para recuperar estos valores iniciales, sino que es posible asignarlos directamente.

6.3.1.3. Definición de las operaciones a probar

A través de la función `command` (la cual recibe como parámetro el estado `S`) es posible definir el conjunto de operaciones que actuarán como transiciones para la máquina de estados. Cada una de las operaciones se especifica a través de una notación simbólica del siguiente tipo:

{call, MODULE, op_i, [$gen\_p_i^1/N_i^1$, $gen\_p_i^2/N_i^2$, ..., $gen\_p_i^{pi}/N_i^{pi}$]}

donde op_i es el nombre de la operación a ejecutar, `MODULE` es el nombre del módulo donde la operación op_i está definida, y $gen\_p_i^j$ es un generador, que recibe N_i^j parámetros, de valores del tipo de dato esperado por el parámetro j de la operación op_i. De esta forma, cada vez que se inicia una nueva prueba, se genera una secuencia aleatoria de llamadas a las operaciones definidas en esta función. Para cada una de ellas, la llamada real que realizará QuickCheck a la hora de ejecutar cada operación es:

MODULE:op_i(P_i^1, P_i^2, ..., P_i^{pi})

donde P_i^j es un valor concreto producido por el generador $gen\_p_i^j$. Por tanto, los argumentos usados en las invocaciones a las operaciones a probar son generados en cada invocación, en vez de usarse valores predefinidos o escritos manualmente.

Como se observa, la especificación de las operaciones se realiza con tuplas con el siguiente formato:

```
{call, Módulo, Función, Argumentos}
```

tratando así las operaciones como una serie de llamadas simbólicas. La razón por la que se realiza esto es que permite a QuickCheck tratar los casos de prueba como estructuras de datos, lo cual es una de las fuentes de sus principales fortalezas [87, 135]. Además, de esta forma, se facilita el depurado cuando se encuentra un error con la ejecución de las pruebas.

Por otra parte, es muy común que el módulo especificado en las llamadas simbólicas sea el propio módulo de pruebas donde se está realizando la definición de la máquina de estados (`?MODULE`). Esto se debe a que este módulo suele definir adaptadores a las funciones reales a ejecutar en el sistema a probar, que permiten adaptar tanto los argumentos de entrada de la operación real a probar, como el resultado devuelto por dicha operación. De esta forma, si la máquina de estados necesita más datos que la operación real no devuelve, algunos de los datos devueltos no se necesitan, o bien si es necesario algún tipo de adaptación de argumentos, definir adaptadores ayuda a mantener el código de la máquina de estados simple y claro, y, obviamente, sin necesidad de modificar el código del sistema a probar. En este trabajo se hará referencia a estas funciones con el nombre de funciones envoltorio.

Por ejemplo, si el sistema a probar proporciona las siguientes operaciones:

$$
\begin{array}{l}
op_1(P_1^1,\ P_1^2,\ \ldots,\ P_1^{p1}) \\
op_2(P_2^1,\ P_2^2,\ \ldots,\ P_2^{p2}) \\
\vdots \\
op_n(P_n^1,\ P_n^2,\ \ldots,\ P_n^{pn})
\end{array}
$$

y dichas operaciones están definidas en el propio módulo de pruebas (`?MODULE`) como funciones envoltorio que invocan las operaciones reales del sistema a probar, la función `command` se escribiría de la siguiente forma:

$$
\begin{array}{l}
\texttt{command(S) ->} \\
\quad \texttt{eqc\_gen:oneof([} \\
\quad\quad \{\texttt{call, MODULE, } op_1\texttt{, [}gen\_p_1^1(S)\texttt{, }gen\_p_1^2(S)\texttt{, }\ldots\texttt{, }gen\_p_1^{pi}(S)\texttt{]}\}\texttt{,} \\
\quad\quad \{\texttt{call, MODULE, } op_2\texttt{, [}gen\_p_2^1(S)\texttt{, }gen\_p_2^2(S)\texttt{, }\ldots\texttt{, }gen\_p_2^{pi}(S)\texttt{]}\}\texttt{,} \\
\quad\quad \vdots \\
\quad\quad \{\texttt{call, MODULE, } op_n\texttt{, [}gen\_p_n^1(S)\texttt{, }gen\_p_n^2(S)\texttt{, }\ldots\texttt{, }gen\_p_n^{pi}(S)\texttt{]}\} \\
\quad \texttt{])}.
\end{array}
$$

donde $gen\_p_i^j$ es un generador de valores del tipo de dato esperado para el parámetro p_i^j. En este ejemplo, los generadores reciben como parámetro el estado `S`, puesto que los valores generados podrían depender de la información almacenada en dicho estado. Aunque esto es bastante común, también podrían no recibirlo o recibir otros parámetros.

Es importante destacar que la función `command` usa el generador `oneof` incluido en QuickCheck, el cual selecciona de forma aleatoria uno de los elementos de la lista que recibe como argumento. En concreto, en este caso, se seleccionarán las operaciones que se ejecutarán en la prueba, conformando así una secuencia de operaciones a ejecutar. Alternativamente, es posible utilizar el generador `frequency`, que permite personalizar la distribución con la que se eligen cada una de dichas operaciones:

```
command(S) ->
 eqc_gen:frequency([
   {F_1, {call, MODULE, op_1, [gen_p_1^1(S), gen_p_1^2(S), ..., gen_p_1^pi(S)]}},
   {F_2, {call, MODULE, op_2, [gen_p_2^1(S), gen_p_2^2(S), ..., gen_p_2^pi(S)]}},
   ⋮
   {F_n, {call, MODULE, op_n, [gen_p_n^1(S), gen_p_n^2(S), ..., gen_p_n^pi(S)]}}
 ]).
```

donde F_i es un número entero no negativo que representa el peso de la operación op_i, y se usa para calcular la probabilidad de seleccionar dicha operación. De esta forma, es posible reflejar mejor una distribución realista de llamadas a operaciones, o incluso todo lo contrario, es decir, forzar que secuencias atípicas se produzcan con mayor facilidad.

Una aproximación recomendable para implementar la función `command` es escribir su implementación de forma progresiva, es decir, incluyendo las operaciones que formarán parte de los comandos de forma gradual. Así, se empezaría generando secuencias de operaciones con pocas transiciones y gradualmente se añadirían más posibilidades.

6.3.1.4. Evolución de la información almacenada en el estado

La máquina de estados QuickCheck requiere definir cómo cada una de las transiciones afecta a la información almacenada en el estado, es decir, cómo la ejecución de cada operación cambia los valores del estado. Esto se realiza a través de la definición de la función `next_state` para cada una de las operaciones especificadas. Esta función recibe como parámetros el estado `S`, el resultado `R` de ejecutar la función envoltorio correspondiente indicada en la función `command` (que como se comentará en la sección 6.3.1.7 puede ser un valor simbólico o un valor real), y la representación simbólica de dicha función envoltorio:

```
next_state(S, R, {call, ?MODULE, op_i, [P_i^1, P_i^2, ..., P_i^pi]}) ->
 #state {
   f1 = next_values_op_i^1(S, R, [P_i^1, P_i^2, ..., P_i^pi]),
   f2 = next_values_op_i^2(S, R, [P_i^1, P_i^2, ..., P_i^pi]),
   ⋮
 };
```

Tanto los datos almacenados en el estado, como la evolución del mismo con la ejecución de cada operación, es una cuestión que depende totalmente del dominio concreto en el que se esté utilizando esta aproximación. En general, esta información depende de las comprobaciones que se deseen realizar en las precondiciones y postcondiciones, puesto que es ahí donde se usarán los valores almacenados en el estado.

6.3.1.5. Precondiciones asociadas a cada operación

Para cada una de las operaciones es posible especificar qué condiciones se deben cumplir para que dicha operación se incluya en una prueba. Estas precondiciones se especifican a través de la función `precondition`, la cual devuelve `true` o `false` en función de la información almacenada en el estado (`S`) y la función envoltorio correspondiente expresada con notación simbólica:

```
precondition(S, {call, ?MODULE, op_i, [P_i^1, P_i^2, ..., P_i^pi]}) ->
 ...;
```

El resultado de esta función es el que va a determinar si una operación se incluirá o no en un punto concreto de la secuencia de operaciones que conforman la prueba, como se explicará en la sección 6.3.1.7.

6.3.1.6. Postcondiciones asociadas a cada operación

Las postcondiciones definen aquellas comprobaciones que deben ser ciertas una vez ejecutada la operación. Estas comprobaciones pueden usar el estado (`S`), los argumentos recibidos por la función envoltorio (P_i^j) y el resultado de la misma (`R`). Las postcondiciones se definen a través de la función `postcondition`, la cual debe devolver `true` si las comprobaciones a realizar se cumplen, o `false` en caso contrario:

```
postcondition(S, {call, ?MODULE, op_i, [P_i^1, P_i^2, ..., P_i^pi]}, R) ->
 ...;
```

Como se explicará en la sección 6.3.1.7, encontrar una postcondición falsa significa encontrar un caso en que el componente no se comporta según lo esperado.

6.3.1.7. Propiedad a comprobar

Por último, es necesario especificar qué propiedad debe ejecutar el sistema de pruebas, en este caso, la herramienta QuickCheck, para comprobar que la implementación del componente se ajusta a lo especificado con la máquina de estados. Esta propiedad suele ser genérica y se suele especificar de una manera similar a esta:

```
prop_state_machine() ->
 ?FORALL(Cmds, eqc_statem:commands(?MODULE),
  begin
    {H, S, Res} = eqc_statem:run_commands(?MODULE, Cmds),
    eqc_statem:pretty_commands(?MODULE, Cmds, {H, S, Res},
       Res == ok)
  end).
```

Esta propiedad indica que para cada secuencia de comandos `Cmds` obtenida a partir del generador de comandos `commands` (el cual usa la función `command` implementada en propio módulo `?MODULE` para generar dichas secuencias), éstas se deben ejecutar, así como comprobar que el resultado de su ejecución se corresponde con lo esperado. Para ello, se debe invocar la función `run_commands`, la cual, además de ejecutar los comandos seleccionados, comprueba las precondiciones y postcondiciones asociadas a cada uno de ellos, además de actualizar el estado después de la ejecución de cada operación. Finalmente, se comprueba que el resultado obtenido `Res` es `ok`, lo cual indica que la secuencia de comandos generada se ha ejecutado de la forma esperada, y las postcondiciones asociadas a cada operación se cumplen.

Es importante destacar que cuando QuickCheck ejecuta esta propiedad, genera cientos de secuencias aleatorias de comandos para, posteriormente, ejecutarlos, es decir, realiza el proceso en dos pasadas, como se muestra en la figura 6.4. Primeramente, genera una secuencia aleatoria de comandos comprobando que las precondiciones especificadas para cada uno de ellos se cumple. En esta primera pasada, QuickCheck no ejecuta las operaciones y, por tanto, trabaja con valores simbólicos para comprobar las precondiciones. Evidentemente, aquellos comandos que no cumplen las precondiciones asociadas no serán incluidos en la secuencia de operaciones a ejecutar. Posteriormente, en una segunda pasada, para cada operación incluida en la secuencia de operaciones generada, QuickCheck comprueba que la precondición se cumple, ejecuta la operación, comprueba que la postcondición se cumple y, finalmente, actualiza el estado usando la función `next_state`.

Si alguna de las precondiciones o postcondiciones no son ciertas en esta segunda pasada, QuickCheck habrá encontrado un contraejemplo, y aplicará un proceso de reducción para encontrar, a partir de la secuencia de operaciones original, la mínima secuencia de comandos que también produzca un fallo [87, 135, 374]. A gran-

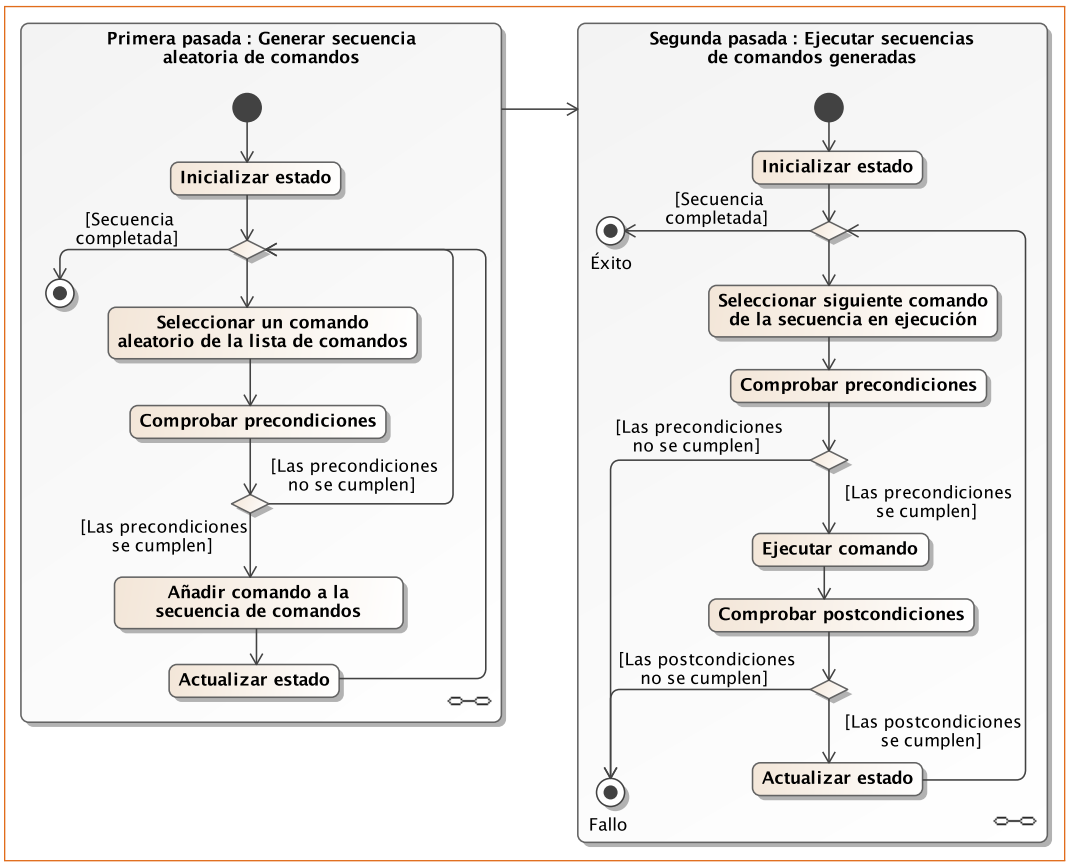


FIGURA 6.4: *Máquina de estados QuickCheck*

des rasgos, el proceso de reducción funciona intentando simplificar los argumentos usados en las llamadas, y también eliminando operaciones de la secuencia de operaciones original (sin que se produzcan nuevas violaciones de las precondiciones), observando si las secuencias de operaciones resultantes son relevantes para producir un fallo. De esta forma, el caso de prueba mínimo resultante de este proceso de reducción ayuda al proceso de depuración para averiguar por qué se ha producido un fallo.

En muchas ocasiones, es útil modificar ligeramente la propiedad para añadir operaciones específicas de inicialización y finalización de las pruebas (a través de la macro `?SETUP`), cambiar el número de veces que se ejecuta la prueba (a través de la función `numtests` de QuickCheck), mostrar más información cuando se encuentra un contraejemplo (usando la macro `?WHENFAIL`), o incluso mostrar estadísticas acerca del porcentaje de ejecución de cada comando (con la función `aggregate`); pero, en cualquier caso, el funcionamiento en dos pasadas es siempre exactamente el mismo.

6.4. Uso de propiedades abstractas para probar APIs de integración

Esta sección describe la aproximación desarrollada para probar APIs de integración [136, 189]. Puesto que una API de integración representa la forma de acceder a un componente desde otros componentes, las implementaciones de éstas se probarán desde un punto de vista externo, es decir, usando una aproximación de pruebas de caja negra. La propuesta que se describe en esta sección, la cual, por tanto, encaja dentro de las pruebas de caja negra, usa la especificación de la API de integración, en vez de la implementación concreta de la misma a probar, para generar, usando una aproximación basada en propiedades, un conjunto de casos de prueba concretos. Puesto que esta aproximación usa la especificación de la API de integración y es independiente de su implementación, las propiedades usadas tampoco dependen de la implementación concreta del sistema a probar y, por tanto, serán denominadas como *propiedades abstractas*.

Por otra parte, la aproximación desarrollada se inspira en la arquitectura utilizada por TTCN-3, puesto que se ajusta perfectamente a las pruebas de APIs de integración. Así, al dividir el código de pruebas en dos partes, ATS y adaptadores que conectan dicho ATS con el sistema a probar, es posible usar la misma especificación de pruebas para probar diferentes implementaciones de la misma especificación de una API de integración.

En este caso, se propone el uso de propiedades para describir la especificación de pruebas, en vez de escribir manualmente un ATS. En concreto, la propuesta se basa en usar máquinas de estados para describir el comportamiento de la API de integración. La especificación de la máquina de estados se extrae de los requisitos del sistema a probar, es decir, de la especificación de la API de integración. A partir de la máquina de estados definida, la herramienta de pruebas basadas en propiedades genera automáticamente casos de prueba, esto es, secuencias aleatorias de llamadas a operaciones de la API de integración a probar.

Este hecho representa una diferencia importante con respecto al uso de TTCN-3, puesto que con la aproximación basada en propiedades se evita escribir uno a uno, de forma manual, los casos de prueba específicos usados en las pruebas, al contrario que en TTCN-3, donde éstos deben ser especificados en uno de los formatos de presentación soportados. Además, el uso de propiedades con respecto a las técnicas de pruebas en las que se especifican los casos de prueba manualmente permite cubrir más casos, y mejora el mantenimiento del código de pruebas cuando el software evoluciona [311].

El resultado de esta propuesta es una metodología de pruebas, especialmente orientada a comprobar el comportamiento de diferentes implementaciones de una misma API de integración. Esta metodología, explicada con detalle en las siguientes secciones, presenta dos ventajas principales:

- El uso de propiedades abstractas evita dependencias con las diferentes implementaciones de una API de integración, minimizando, por tanto, el esfuerzo de probar los requisitos funcionales y la equivalencia funcional entre ellos en las diferentes implementaciones.
- El mantenimiento de las pruebas, es decir, de las propiedades que describen el sistema a probar, se simplifica. Además de escribir sólo una especificación de pruebas por cada especificación de la API de integración a probar (en vez de una por cada implementación concreta a probar), no todos los cambios en la implementación del sistema a probar implican cambios en el código de especificación de pruebas. Así, sólo se actualizarán las propiedades si la especificación de la API de integración cambia.
- Además, al usar una aproximación basada en propiedades, es importante mencionar que no es necesario realizar ningún tipo de mantenimiento sobre los casos de prueba, sino únicamente sobre la especificación de las pruebas (es decir, las propiedades), puestos que éstos se generan automáticamente cuando las pruebas se ejecutan a partir de las propiedades definidas.

A continuación se presenta una implementación de esta metodología usando la versión Erlang de QuickCheck, que permite generar automáticamente casos de prueba a partir de propiedades o modelos abstractos. Así, en vez de TTCN-3, esta propuesta usa Erlang como lenguaje de especificación de pruebas, independientemente del lenguaje en el que el sistema a probar esté implementado. Esto es posible puesto que para comprobar si las propiedades definidas son ciertas, es decir, para ejecutar los casos de prueba generados a partir de las mismas, se usa el mismo concepto de abstracción definido por TTCN-3 (ver figura 6.1 de la página 115).

6.4.1. Metodología de pruebas

La figura 6.5 muestra un diagrama con las actividades que forman parte de la metodología propuesta. Estas actividades son las que se detallan a continuación.

6.4.1.1. Especificación de requisitos

Los requisitos representan las necesidades que el sistema o componente a ser implementado debe satisfacer. Existen diferentes tipos de requisitos: de funcionalidad, de rendimiento, de consumo de recursos, etc. Esta propuesta se basa únicamente en los requisitos funcionales para producir casos de prueba que permitan comprobar el comportamiento funcional del sistema a probar.

Cuando el sistema a probar es una implementación de una API de integración, esta propuesta se basa en la especificación de dicha API. Dicha especificación puede ser formulada de forma abstracta de la siguiente manera:

$$API = \{op_1, op_2, \ldots, op_o\}$$

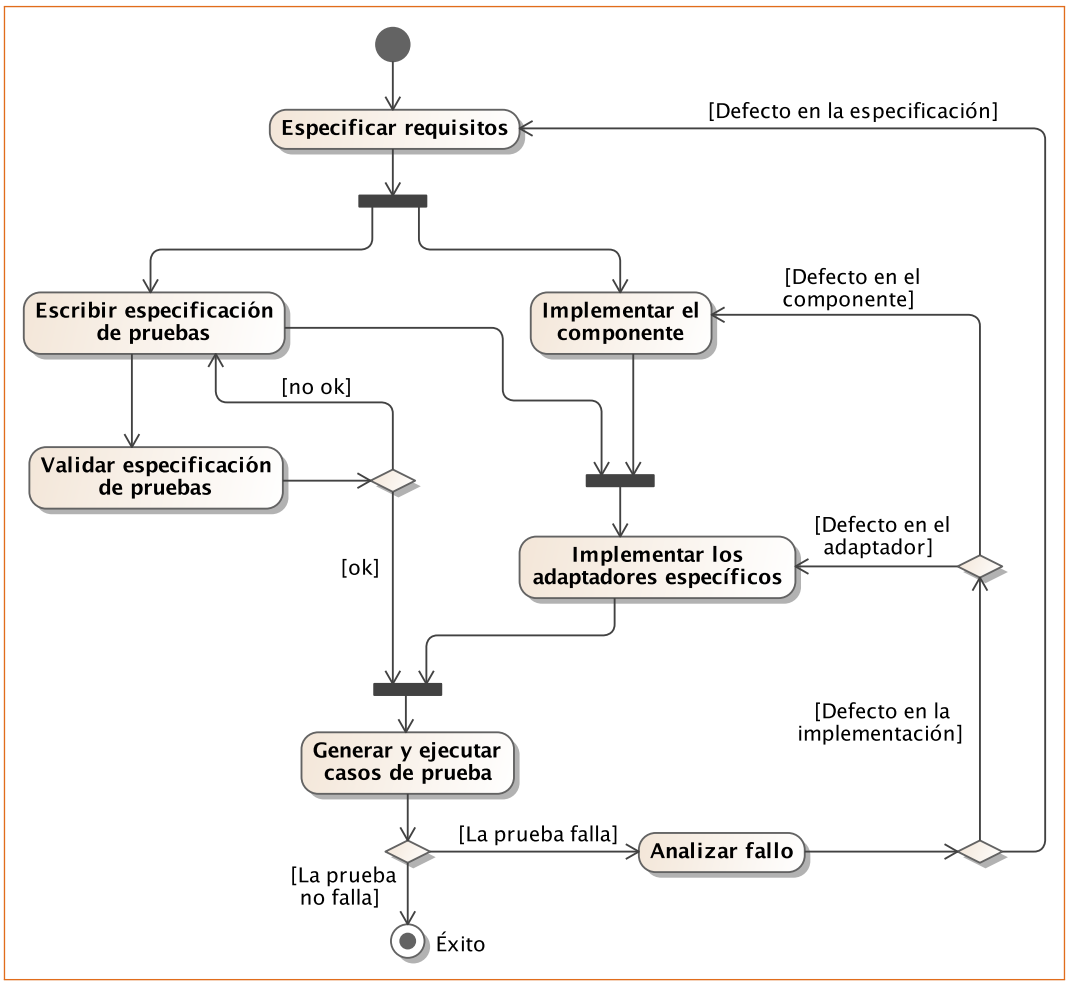


FIGURA 6.5: *Metodología desarrollada para probar APIs de integración*

donde o es el número de operaciones proporcionadas por la API de integración, y op_i representa la operación i de dicha API. A su vez, cada operación se define de la siguiente forma:

$$op_i = \{n_i, p_i, r_i, e_i, c_i\}$$

donde:

- n_i es el nombre de la operación,
- p_i representa los parámetros que recibe la operación, es decir, $\{p_i^1, p_i^2, \ldots, p_i^p\}$. Cada parámetro $p_i^j = \{pn_i^j, pd_i^j\}$ es a su vez representado por su nombre (pn_i^j) y su tipo de dato (pd_i^j),
- r_i es el tipo de dato que devuelve la operación,
- e_i representa los posibles errores que podría devolver la operación, es decir,

$\{e_i^1, e_i^2, \ldots, e_i^e\}$,

- c_i representa las restricciones $\{c_i^1, c_i^2, \ldots, c_i^c\}$ asociadas a la operación, es decir, las condiciones asociadas a la ejecución de la operación, tanto las que deben ser ciertas antes de poder invocar la operación, como las que se deben cumplir después de su ejecución. Estas restricciones pueden depender de los parámetros p_i que recibe la operación, e incluso de otras operaciones invocadas anteriormente, es decir, del estado del sistema o del componente.

Con esta especificación, además de las operaciones que componen la API de integración, junto con sus parámetros de entrada, tipo de dato de retorno, y los posibles errores asociados; se especifican cuándo se deben devolver dichos valores de retorno o cuándo se deben devolver errores a través de las restricciones asociadas c_i. De esta forma, se define una especificación completa de la API de integración que se usará para generar, bien manualmente o bien automáticamente, los casos de prueba que comprobarán el funcionamiento de las implementaciones de dicha API de integración.

Como se explicará en la sección 6.5, esta especificación puede ser formalizada usando algún tipo de lenguaje de modelado, como son UML y OCL, que permita representar tanto las operaciones, junto con sus parámetros y tipos de datos de retorno, como las restricciones asociadas como parte de los requisitos.

6.4.1.2. Especificación de pruebas

Después de la especificación de requisitos, el siguiente paso es escribir la especificación de pruebas a partir de la cual los casos de prueba puedan ser generados para probar dichos requisitos. Dicha especificación de pruebas es una especificación *abstracta*, puesto que no depende de la implementación específica a probar, sino que se obtiene a partir de la especificación de requisitos y, por tanto, puede ser utilizada para probar todas las implementaciones de dicha especificación. Este hecho implica que esta aproximación requiera una arquitectura más compleja que las aproximaciones en las que las pruebas son escritas teniendo en cuenta directamente la implementación a probar.

Siguiendo una aproximación basada en propiedades, los requisitos se transforman en propiedades que se usarán para probar el software. En este caso, al tratarse el componente a probar de una API de integración, se usará una máquina de estados, usando QuickCheck como herramienta de pruebas. La máquina de estados se adapta perfectamente al concepto de pruebas de caja negra de APIs de integración, puesto que éstas están compuestas por una serie de operaciones relacionadas, las cuales tienen una serie de precondiciones y postcondiciones asociadas. Por tanto, el objetivo principal de esta fase es traducir la especificación de requisitos $API = \{op_1, op_2, \ldots, op_o\}$ a una máquina de estados QuickCheck.

El primer paso para construir la máquina de estados es identificar los comandos que forman parte de la misma. En este caso, dichos comandos son las operaciones op_i que componen la API de integración a probar. Siguiendo la recomendación explicada en la sección 6.3, para cada operación op_i se definirá una función envoltorio $op'_i = \{n'_i, p'_i, r'_i, e'_i, c'_i\}$ que invocará la propia operación a probar op_i. Por otra parte, es necesario definir qué información se debe almacenar en el estado para que las precondiciones y postcondiciones asociadas a cada operación puedan ser comprobadas. Finalmente, las precondiciones, postcondiciones y funciones de modificación de estado (`next_state`) también deben ser implementadas. Para esto, se deben seguir los pasos explicados en la sección 6.3.1.

Puesto que la especificación de la máquina de estados no depende de la implementación concreta del sistema a probar, los objetos del dominio que formen parte de dicho sistema a probar tendrán una representación abstracta en la especificación de pruebas. Al ser Erlang el lenguaje usado en la especificación de pruebas, una posible representación de estos objetos podría estar basada en el uso de una lista de tuplas (conocida como *lista de propiedades* o *proplist* en Erlang) donde cada tupla representa un atributo del objeto del dominio con su valor. Por ejemplo, dado un objeto y del tipo Y con los atributos a_1, a_2, ..., a_x, con valores v_1, v_2, ..., v_x, la representación abstracta de dicho objeto podría ser:

$$\{Y, [\{a_1, v_1\}, \{a_2, v_2\}, \ldots, \{a_x, v_x\}]\}$$

donde v_i podría ser, a su vez, un objeto complejo, es decir, que contiene sus propios atributos, que, a su vez, se representaría con una nueva tupla. Sin embargo, es responsabilidad de la persona que escribe dicha especificación la selección de la estructura abstracta de dichos objetos. Como se explicará posteriormente, existirá un adaptador (que no formará parte de la especificación de pruebas, pero sí del código de pruebas) que se encargará de transformar esta representación abstracta en la especificación requerida por la implementación concreta a probar, y viceversa.

De esta forma, después de escribir la máquina de estados QuickCheck, existirá una especificación de pruebas completa, independiente de las implementaciones concretas de la API de integración a probar. Esta especificación la utiliza la propia herramienta QuickCheck para generar casos de prueba que permitan comprobar si una implementación específica de una API de integración satisface la especificación de dicha API.

En resumen, los requisitos funcionales, representados por las operaciones op_i, se transforman en funciones envoltorio op'_i. Además, las restricciones asociadas a cada operación c'_i se formalizan como una serie de precondiciones y postcondiciones que se incluyen en la especificación de pruebas. Esta abstracción y formalización puede ser realizada completamente por un probador de software, es decir, una persona (o grupo de personas) que podría ser diferente de los programadores que escriben las diferentes implementaciones de la API de integración. Este proceso, como se indica

en la figura 6.5, puede comenzar incluso antes que la propia implementación del componente, y también puede, por tanto, realizarse de forma paralela y sin interferir con dicha implementación.

6.4.1.3. Validación de la especificación de pruebas

Escribir propiedades es un proceso en el cual se pueden producir errores, de la misma forma que puede suceder al escribir cualquier otro tipo de especificación de pruebas, y naturalmente al igual que al implementar el software en sí mismo. Es por ello que es importante validar que las propiedades escritas son realmente adecuadas para probar el funcionamiento del software.

Puesto que, como se muestra en la figura 6.5, el componente a probar no necesita estar disponible para comenzar a definir los casos de prueba, se propone que los probadores de software usen un componente de reemplazo [264]. El uso de componentes de reemplazo es muy común en las actividades de pruebas. Como se explicará en la sección 8.2 del capítulo 8, existen diferentes tipos de componentes de reemplazo (componentes *dummies*, *stubs*, *espías*, *mocks*, o componentes *ficticios*) dependiendo de la "inteligencia" que tengan. En las pruebas de unidad, este tipo de componentes son usados típicamente para reemplazar sistemas externos por implementaciones controladas de dichos componentes [212, 254, 264]. Por otro lado, en las pruebas de integración, además de reemplazar componentes externos, también son usados para interceptar invocaciones a operaciones de dichos componentes reemplazados, y compararlas con la traza esperada de invocaciones [138].

En este caso, la idea propuesta es usar un componente *ficticio*, que simule el comportamiento del componente real. Los componentes *ficticios* representan una implementación alternativa de un componente real, ofreciendo la misma API de integración que éste (al menos, la parte necesaria para las pruebas), y proporcionando respuestas para las operaciones que forman parte de la misma. De esta forma, aunque el componente real no esté disponible, los casos de prueba pueden ser ejecutados igualmente como si lo estuviera, aunque evidentemente el funcionamiento global del sistema no va a ser el esperado, puesto que el componente ficticio no realizará todas las acciones que sí realiza el componente real reemplazado. En cualquier caso, este componente *ficticio* puede ser usado para refinar la especificación de pruebas de forma iterativa.

La implementación de estos componentes de reemplazo debería ser una tarea fácil en la mayoría de los casos, puesto que no deben ser un modelo completo del componente real a reemplazar, sino que únicamente se necesita que funcionen de la misma manera que el componente original desde un punto de externo, es decir, ofreciendo las mismas operaciones que la API de integración a probar. Así, cada operación reemplazada debe aceptar exactamente los mismos parámetros y retornar el mismo tipo de valores que la correspondiente operación en el componente original. En este caso, las operaciones de la API de integración podrían retornar, por

ejemplo, simplemente valores fijos, o incluso incluir algún tipo de lógica básica que pueda ayudar a comprobar los efectos colaterales en la ejecución de las operaciones (como, por ejemplo, encontrar un objeto que fue previamente creado).

En cualquier caso, es importante enfatizar que el componente ficticio no es el componente real. Incluso aunque su intención es mimetizar el comportamiento del componente original, su único propósito es ayudar a validar la especificación de pruebas.

6.4.1.4. Implementación del componente

Mientras que el equipo de probadores de software escribe la especificación de pruebas que servirá para generar automáticamente los casos de prueba, siempre desde una perspectiva de caja negra, sin tener en cuenta los detalles de bajo nivel del sistema a probar; el equipo de programadores se centrará en la propia implementación del componente (ver figura 6.5).

Este componente a implementar debe, obviamente, satisfacer los requisitos establecidos y, además, las personas que se encarguen de su desarrollo deben comprobar dicha implementación desde un punto de vista interno, es decir, implementando y ejecutando pruebas de unidad. Para esto, es posible usar las técnicas explicadas en el capítulo 5.

6.4.1.5. Implementación de los adaptadores específicos

Una vez definida la máquina de estados QuickCheck, es decir, la especificación de pruebas a partir de la cual los casos de prueba serán automáticamente generados; y cuando exista una primera implementación del componente real (o bien de algún componente ficticio que reemplace al real implementando su API de integración), el siguiente paso es conectar dicha especificación de pruebas abstracta, en concreto, los casos de prueba abstractos generados por dicha especificación (ATS), con el componente a probar.

Como se comentó en la sección 6.2.2, existen varias aproximaciones para conectar un ATS con el sistema a probar [348], como son el uso de un adaptador, la transformación de los casos de prueba abstractos en casos de prueba concretos a través de *scripts*, o aproximaciones que combinan ambas técnicas. En este caso, tomando la arquitectura de TTCN-3 como inspiración, se ha elegido usar una capa de adaptación para conectar ambas partes, como se muestra en el diagrama de la figura 6.6. Es por ello que el uso de esta aproximación requiere la implementación de adaptadores específicos que se encarguen de realizar las llamadas concretas al sistema a probar.

Como se muestra en la figura 6.5, no es necesario haber validado completamente la especificación de pruebas para comenzar con la implementación del adaptador

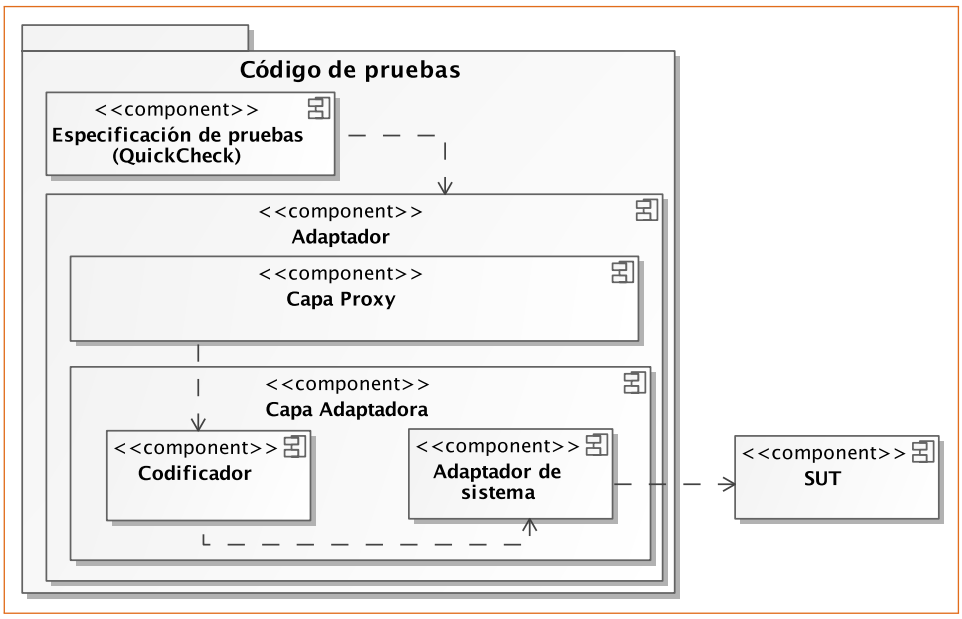


FIGURA 6.6: *Arquitectura propuesta para probar APIs de integración*

(o adaptadores). La razón es que para construir el adaptador sólo es necesario conocer las operaciones a probar, es decir, su sintaxis y tipos de datos, para así poder implementar el proceso de codificación y decodificación que debe realizar dicho adaptador. Obviamente, los casos de prueba no serán ejecutados mientras que la especificación de pruebas y los adaptadores no estén implementados.

En la aproximación propuesta se ha decidido añadir una capa más a la capa de adaptación, a la que se llama *proxy* (ver figura 6.6). Esta capa se encarga de conectar las funciones envoltorio empleadas en la máquina de estados QuickCheck, es decir, las funciones op'_i, con el adaptador propiamente dicho. Esta capa es útil cuando la especificación de pruebas y el adaptador están escritos en lenguajes de programación diferentes.

Por su parte, la subcapa adaptadora propiamente dicha se encarga de invocar las operaciones específicas en el sistema a probar, transformando los datos que proceden de la especificación de pruebas a un formato que dicho sistema entienda, y recibiendo los datos del sistema a probar para procesarlos y enviarlos de vuelta a la implementación de la especificación de pruebas en el formato correspondiente. La arquitectura de esta subcapa se inspira en TTCN-3 y, de esta forma, se divide en dos partes: la subcapa de codificación y la subcapa de adaptación de sistema. Sin embargo, tal y como se explicará posteriormente con un caso de estudio (en la sección 6.4.2), la arquitectura de esta subcapa adaptadora puede ser simplificada notablemente cuando la especificación de pruebas y el sistema a probar están escritos en el mismo lenguaje de programación, es decir, Erlang.

6.4.1.6. Generación y ejecución de los casos de prueba

Una de las últimas fases en la metodología propuesta (ver figura 6.5) es la generación y ejecución de los casos de prueba. Para ello, debe tenerse en cuenta que el uso de QuickCheck para escribir propiedades que el software debe satisfacer sirve no sólo para generar automáticamente casos de prueba, sino también para ejecutarlos.

En este caso, al tratarse la especificación de pruebas de una máquina de estados, se generarán y ejecutarán múltiples secuencias de operaciones aleatorias, todas ellas combinaciones de las operaciones listadas en la función `command` de la máquina de estados (ver sección 6.3.1). Además, los argumentos que recibe cada operación también son generados por QuickCheck. Esto supone una diferencia importante con respecto a las aproximaciones en las que tanto la secuencia de operaciones, como los argumentos que recibe cada una de ellas, suelen ser valores especificados manualmente en el código de pruebas.

Durante la generación y ejecución de las operaciones, QuickCheck comprueba las precondiciones y postcondiciones asociadas a cada una de ellas. Como se ya ha explicado anteriormente, dichas precondiciones y postcondiciones se corresponden con las restricciones asociadas a las operaciones de la API de integración especificadas en los requisitos iniciales del sistema.

Las secuencias de operaciones generadas, con los argumentos concretos, son entonces ejecutadas, en este caso, resultando en llamadas a la capa de adaptación, la cual que se encargará de invocar la implementación concreta de la API de integración a probar.

6.4.1.7. Análisis de los resultados de las pruebas

Como se muestra en la figura 6.5, si la ejecución de los casos de prueba revela algún tipo de error, el origen de éste debe ser analizado. Al usar QuickCheck para ejecutar las pruebas, cuando se encuentra un error, QuickCheck mostrará el caso de prueba que lo provoca, es decir, la secuencia de operaciones que lleva a la violación de una postcondición. Además, para ayudar a la búsqueda y diagnóstico del problema, QuickCheck aplicará un proceso de reducción que se encargará de buscar, a partir del caso de prueba original que causa un comportamiento erróneo, un caso de prueba más pequeño, esto es, una secuencia de prueba más corta y con argumentos estructuralmente más simples (lo que generalmente se traduce en valores “menores”, y estructuras de datos más “cortas” o con menos elementos), que también viole una postcondición [87, 135, 374].

Aunque se espera que los defectos aparezcan en el sistema a probar, éstos también podrían aparecer en la especificación de pruebas o incluso en el código del adaptador (o adaptadores). Cuando los defectos se encuentran en el sistema a probar, que es el objetivo principal de cualquier estrategia de pruebas, evidentemente éstos se deben solucionar modificando el propio sistema a probar.

Por otro lado, los defectos pueden aparecer también en el adaptador, bien la primera vez que se implementa para conectar el ATS con una implementación específica a probar, o incluso después de que ciertas partes de un adaptador existente no fuesen extensivamente ejecutadas (o probadas) durante su uso previo. A pesar de esto, el potencial que supone poder reusar la especificación de pruebas convierte al adaptador en una pieza relevante de la arquitectura propuesta.

Finalmente, las pruebas también podrían fallar porque la especificación de pruebas es incorrecta. Así, es posible que se hayan introducido defectos al escribir las propiedades, o bien haya ambigüedades en los requisitos, requisitos no especificados, o requisitos sin una especificación adecuada. En este caso, la especificación de pruebas debe ser corregida, y el proceso de generación de los casos de prueba debe ser comenzado de nuevo.

6.4.1.8. Incorporación de nuevos requisitos al sistema

Cuando aparecen nuevos requisitos a implementar, cada uno de los pasos descritos en la figura 6.5 debería ser repetido de nuevo. Esto incluye añadir nuevas propiedades a la especificación de pruebas o modificar las propiedades ya existentes si los requisitos previos han sido cambiados; así como modificar el adaptador para tener en cuenta las nuevas operaciones y las nuevas entidades a codificar y decodificar.

En este proceso, usar un componente ficticio que simule el comportamiento del sistema a probar puede ayudar a los probadores de software en la tarea de modificar la especificación de pruebas para satisfacer los nuevos requisitos, o las modificaciones de los ya existentes. Mientras tanto, los programadores pueden centrarse en la propia implementación de los nuevos requisitos en el propio sistema a probar.

Posteriormente, un nuevo conjunto de casos de prueba automáticamente generados se ejecutará contra el componente actualizado usando la capa de adaptación correspondiente. Finalmente, si aparecen errores en este punto, éstos deben ser analizados para poder ser solucionados.

6.4.2. Caso de estudio: componente de gestión de contenidos multimedia

Esta sección ilustra cómo se ha aplicado la metodología propuesta en la implementación de un sistema real: la API de integración de VoDKA Asset Manager. Este componente es un gestor de metainformación de contenidos multimedia, los cuales se conocen como *assets* en la nomenclatura de este componente, y cuyos medios físicos asociados suelen estar almacenados en un almacén de medios gestionados por el servidor de vídeo *VoDKA* [200].

La metainformación almacenada por el componente VoDKA Asset Manager se compone, entre otros campos, de título, resumen, género, fecha, director, actores, o clasificación de control parental. Además, aparte de almacenar metainformación,

VoDKA Asset Manager también ofrece búsquedas de contenidos (*assets*) usando diferentes criterios, actualización de la metainformación asociada a contenidos existentes y borrado de contenidos, entre otras funcionalidades.

El componente VoDKA Asset Manager está siendo usado actualmente como aplicación de *back-end* en varios despliegues reales. Así, como se muestra en la figura 3.6 de la página 56, VoDKA Asset Manager es parte de la arquitectura de VoDKATV, donde es usado para almacenar metainformación asociada a contenidos multimedia que los usuarios finales pueden alquilar y reproducir en sus televisiones (utilizando un *set-top-box*), tabletas, teléfonos móviles, o cualquier otro dispositivo compatible, utilizando para ello algún tipo de aplicación cliente que permita mostrar esta información y reproducir los contenidos multimedia correspondientes. Además, existen despliegues de Internet televisión, en los cuales los usuarios finales pueden reproducir contenidos multimedia usando un navegador web estándar (Firefox, Internet Explorer, Chrome, etc.), que usan VoDKA Asset Manager de la misma manera que en los despliegues IPTV y OTT. Por otra parte, VoDKA Asset Manager también es vendido de forma separada por Interoud Innovation para ser usado como gestor de contenidos externo para aplicaciones de terceros.

En resumen, VoDKA Asset Manager es usado por otros componentes (generalmente aplicaciones *front-end*) para recuperar y actualizar metainformación asociada a contenidos multimedia. Como estos componentes pueden ser muy heterogéneos, VoDKA Asset Manager ofrece una API de integración adaptada para poder ser usada desde diferentes tecnologías, todas ellas ofreciendo las mismas funcionalidades. En concreto, VoDKA Asset Manager incorpora una librería Java (JavaBali) para ser usada desde aplicaciones Java, una aplicación Erlang (ErlBali) para poder ser invocada desde código Erlang, y un servicio web genérico HTTP/XML (ver figura 6.7). Puesto que la funcionalidad que se ofrece es la misma desde cualquiera de estas implementaciones de la API de integración, tiene sentido que, a la hora de probar estas implementaciones, el conjunto de casos de prueba para cada implementación pueda ser automáticamente generado a partir de una especificación de pruebas común.

Con la aproximación propuesta, las tres implementaciones de la API de integración se probarán con la misma especificación de pruebas, la cual describe el comportamiento de dicha API. De esta forma, con una única especificación, compuesta por propiedades, obtenida a partir de la especificación de la API de integración, es posible generar automáticamente casos de prueba válidos para probar todas las implementaciones de dicha API de integración. Así, simplemente usando adaptadores específicos para cada una de las tecnologías (Java, Erlang y HTTP/XML), es suficiente para probar las tres implementaciones concretas de la API de integración. De esta forma, se valida que la arquitectura propuesta por TTCN-3 se adapta perfectamente a esta aproximación.

A continuación se describen cada uno de los pasos de esta metodología aplicados al caso de estudio específico de VoDKA Asset Manager.

[1] Se han omitido algunos de los atributos asociados a un `asset` (control parental, carátula, palabras clave, categorías, etc.) para mantener la simplicidad.

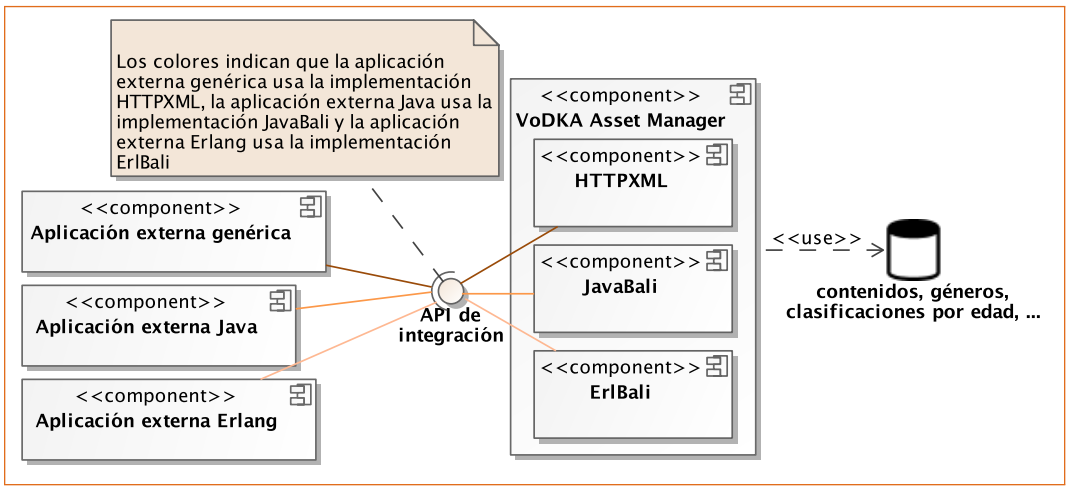


FIGURA 6.7: *Arquitectura de VoDKA Asset Manager*

6.4.2.1. Especificación de requisitos

La API de integración de VoDKA Asset Manager ofrece operaciones para administrar la metainformación de los contenidos multimedia almacenados, es decir, los *assets*, y los atributos compuestos asociados (géneros, categorías, clasificaciones por edad, etc.). Por cada contenido, VoDKA Asset Manager permite almacenar, entre otros datos, la siguiente información[1] :

- `assetId`: valor alfanumérico no vacío que identifica unívocamente al contenido.
- `title`: título del contenido.
- `summary`: descripción asociada al contenido.
- `creationDate`: fecha de creación del contenido expresada como el número de milisegundos transcurridos desde el 1 de enero de 1970 hasta dicha fecha.
- `runtime`: duración del contenido en el formato `hh:mm:ss`.

La tabla 6.1 enumera algunas de las operaciones de VoDKA Asset Manager relacionadas con la gestión de contenidos, además de las operaciones para autenticarse en el sistema y cerrar la sesión en el mismo. Se omiten, por simplicidad, otras operaciones que forman parte de la misma API de integración. Como regla general, el flujo de operaciones para acceder a VoDKA Asset Manager incluye una llamada de autenticación al sistema usando la operación `init` con un usuario (parámetro `login`) y contraseña (parámetro `password`) válidos. Una vez autenticado en el sistema, es posible invocar las operaciones para administrar los contenidos (`create`, `find_by_id`, `find_all`, `update` o `delete`). Finalmente, para cerrar la sesión en el sistema, se debe invocar la operación `reset`.

Operación (n_i)	**Parámetros** (p_i)	**Resultado** (r_i)	**Errores** (e_i)

`create`	asset: `asset`	`ok`	`duplicated_asset`
			`not_started`
			`authentication_error`
			`mandatory_field_id`
			`connection_error`

`find_by_id`	id: `id`	`asset`	`not_found`
			`not_started`
			`authentication_error`
			`connection_error`

`find_all`	—	`list <asset>`	`not_started`
			`authentication_error`
			`connection_error`

`update`	asset: `asset`	`ok`	`not_found`
			`not_started`
			`authentication_error`
			`connection_error`

`delete`	id: `id`	`ok`	`not_found`
			`not_started`
			`authentication_error`
			`connection_error`

`init`	url: `url` login: `string` password: `string`	`ok`	`already_started`
`reset`	—	`ok`	`not_started`

TABLA 6.1: *Especificación de la API de integración de VoDKA Asset Manager*

En cualquier caso, incluso aunque la tabla 6.1 especifica todas las respuestas y errores posibles para cada operación, se necesita una especificación completa del protocolo de comunicación para poder definir los casos de prueba que permitan probar las implementaciones de esta API de integración. Esta especificación debería incluir las condiciones en las cuales se obtienen cada una de las respuestas y errores, es decir, las restricciones asociadas a cada operación c_i. Por ejemplo, para la operación `create`, dichas restricciones serían las siguientes:

- La operación puede ser ejecutada bajo cualquier circunstancia y en cualquier momento, es decir, no tiene precondiciones asociadas.
- El resultado de ejecutar la operación es el error `not_started` si anteriormente no se ha realizado una llamada a la operación `init`.

- El resultado de ejecutar la operación es el error `connection_error` si el servidor VoDKA Asset Manager anteriormente inicializado con la operación `init` no está disponible.
- El resultado de ejecutar la operación es el error `authentication_error` si la combinación de `login` y `password` usados en la llamada a la operación `init` es incorrecta.
- El resultado de ejecutar la operación es `mandatory_field_id` si el identificador del `asset` a crear no se especifica.
- El resultado de ejecutar la operación es el error `duplicated_asset` si el valor del parámetro `asset` contiene un valor para el campo `assetId` perteneciente a un contenido ya existente en el sistema.
- Por último, la operación devuelve `ok` si el `asset` ha podido ser creado satisfactoriamente en el sistema.

Aunque esta especificación puede ser formalizada usando algún tipo de lenguaje, como se mostrará en la sección 6.5, donde se usa el lenguaje de modelado UML junto con restricciones OCL para este fin, éste no es el objetivo de este caso de estudio, por lo que únicamente se mostrará cómo transformar esta especificación, escrita en lenguaje natural, en propiedades que permitan generar casos de prueba que se usarán para probar las implementaciones de esta API de integración.

6.4.2.2. Especificación de pruebas

Una vez terminada la especificación de los requisitos de la API de integración, es posible escribir la máquina de estados QuickCheck que va a permitir comprobar si éstos han sido implementados de la manera deseada, usando una aproximación basada en propiedades. Los comandos de la máquina de estados serán las funciones envoltorio op'_i que llamarán a las operaciones reales op_i. Así, en este caso, la función `command`, donde se enumeran estas operaciones, se podría escribir de la siguiente manera:

```
command(S) ->
 eqc_gen:oneof([{call, ?MODULE, create, [gen_asset()]},
      {call, ?MODULE, find_all, []},
      {call, ?MODULE, find_by_id, [gen_valid_id()]},
      {call, ?MODULE, update, [gen_asset()]},
      {call, ?MODULE, delete, [gen_valid_id()]},
      {call, ?MODULE, init, [gen_url(), gen_user_name(),
                     gen_password()]},
      {call, ?MODULE, reset, []}]).
```

donde `?MODULE` es el propio módulo de pruebas.

Además, como se observa, los argumentos usados en las invocaciones de las funciones op'_i se generan aleatoriamente en vez de usar valores concretos. De esta forma, las funciones `gen_asset`, `gen_valid_id`, `gen_url`, `gen_user_name` y `gen_password` son generadores de datos implementados por quien haga la especificación, trasladando los requisitos sobre los datos de entrada aceptables, que generan `assets`, identificadores de `assets`, URLs, nombres de usuarios y contraseñas, respectivamente.

Por otro lado, al definir la especificación de pruebas de una forma abstracta, independiente de la implementación, es necesario también definir la estructura de los objetos del dominio, en este caso, de los `assets`. Si se usa la estructura explicada en la descripción de esta metodología, en la que se usa una tupla cuyo segundo elemento es una lista de tuplas con los atributos correspondientes, una representación abstracta de un `asset` es:

```
{asset, [{assetId, <ID>}, {title, <TIT>}, {summary, <SUM>},
 {creationDate,<CDATE>}, {runtime, <RTIME>}]}
```

donde `<ID>` es el valor del atributo `assetId`, `<TIT>` el valor del atributo `title`, `<SUM>` el valor del atributo `summary`, `<CDATE>` representa el valor del atributo `creationDate`, y `<RUN_TIME>` es el valor del atributo `runtime`. Esta estructura abstracta será transformada en una representación concreta, y viceversa, por el adaptador empleado en las pruebas.

Por tanto, los generadores producirán datos teniendo en cuenta esta estructura abstracta. Así, por ejemplo, la implementación del generador `gen_asset` es la siguiente:

```
gen_asset()->
 {asset,
  [{assetId, gen_valid_id()},
   {title, gen_string()},
   {summary, gen_string()},
   {creationDate,
     ?LET(X, eqc_gen:choose(seconds({1900,1,1}),
                            seconds({2200,1,1})),
          integer_to_list(X))},
   {runtime, ?LET(T, eqc_gen:nat(), format_runtime(T))}]
 }.
```

donde `gen_valid_id` y `gen_string` son, a su vez, generadores de datos que producen identificadores válidos y cadenas de caracteres respectivamente; la función `choose` es un generador incluido con QuickCheck, que recibe dos parámetros `R1` y `R2`, de tal forma que elige un valor aleatorio en un rango `[R1,...,R2]`; `nat` es otro generador que proporciona QuickCheck para generar números naturales; `seconds` es una función que devuelve el número de segundos desde 1970 de una fecha deter-

minada; y `format_runtime` es función que transforma un número natural `T` que recibe como parámetro en una cadena de caracteres con formato `hh:mm:ss` (donde `hh` es el número de horas, `mm` los minutos y `ss` los segundos) que representa la duración de un contenido.

Así, con el generador `gen_asset` se generan valores como los siguientes (obtenidos con la función `sample` que ofrece QuickCheck en el módulo `eqc_gen`):

```
{asset, [{assetId,"to"}, {title,"Oe"}, {summary,"r"},
 {creationDate,"61469380303"}, {runtime,"0:00:01"}]}

{asset, [{assetId,"n"}, {title,"fUF"}, {summary,"lKg"},
 {creationDate,"60266342370"}, {runtime,"0:00:01"}]}

{asset, [{assetId,"K"}, {title,"sVO"}, {summary,"it"},
 {creationDate,"67253779811"}, {runtime,"0:00:08"}]}

{asset, [{assetId,"Qw"}, {title,"k"}, {summary,"E"},
 {creationDate,"68184792270"}, {runtime,"0:00:00"}]}

{asset, [{assetId,"rg"}, {title,"lIQ"}, {summary,"JeDLP"},
 {creationDate,"69334239484"}, {runtime,"0:00:03"}]}

{asset, [{assetId,"w"}, {title,"XRb"}, {summary,"EsiKz"},
 {creationDate,"67863908653"}, {runtime,"0:00:03"}]}

{asset, [{assetId,"DSX"}, {title,"AzFmH"}, {summary,"JY"},
 {creationDate,"62204329261"}, {runtime,"0:00:16"}]}

{asset, [{assetId,"iHB"}, {title,"DnwUEH"}, {summary,"bjMZiN"},
 {creationDate,"69003543498"}, {runtime,"0:00:14"}]}

{asset, [{assetId,"KfW"}, {title,"xQA"}, {summary,"GAcdPv"},
 {creationDate,"67473242295"}, {runtime,"0:00:08"}]}

{asset, [{assetId,"KobH"}, {title,"lkVD"}, {summary,"NvZjrVP"},
 {creationDate,"60441487410"}, {runtime,"0:00:04"}]}

{asset, [{assetId,"ZCvuNzM"}, {title,"nRJZ"}, {summary,"oY"},
 {creationDate,"66476453899"}, {runtime,"0:00:16"}]}
```

Por otro lado, en este caso, además de las operaciones que forman parte de la API de integración, es necesario identificar qué información debe ser almacenada en el estado de la máquina de estados para poder escribir la especificación de pruebas. Esta información, que varía según el sistema concreto a probar, será usada tanto por

los generadores como por las postcondiciones de cada una de las operaciones. Normalmente, se suelen almacenar todos aquellos objetos del dominio que son creados durante las pruebas, y que se necesitan recuperar durante las mismas, y, en general, toda la información necesaria durante una prueba para comprobar que las operaciones se han ejecutado de la manera esperada. En este caso, además de la lista de contenidos creados en el sistema, es decir, los `assets`, también se almacenará información acerca de la inicialización de la API de integración usando la operación `init`, en concreto:

- `started`: indica si la operación `init` ha sido invocada, y la operación `reset` no ha sido llamada después.
- `valid_url`: indica que la operación `init` ha sido invocada con una URL válida (sin llamar a `reset` posteriormente).
- `valid_login`: indica que la operación `init` ha sido invocada con una combinación de `login` y `password` válida (sin llamar a `reset` posteriormente).
- `assets`: lista de `assets` creados en el sistema.

Esta información se almacenará en un registro Erlang con estos campos:

```
-record(state, {started, valid_url, valid_login, assets}).
```

Además, es necesario indicar cómo se inicializa dicho registro a través de la función `initial_state`. En este caso, ésta será la inicialización:

```
initial_state() ->
   #state {
      started = false,
      valid_url = false,
      valid_login = false,
      assets = []
   }.
```

La información almacenada en el estado puede ser usada para mejorar los generadores. Por ejemplo, la operación `create` (al igual que otras operaciones) devuelve resultados diferentes en función de si el `asset` usado como argumento ya existe en el sistema o no. En este caso, si el `asset` ya existe, la operación `create` devolverá un error `duplicated_asset`, mientras que si no existe y se crea satisfactoriamente, se devolverá como respuesta `ok`. Al usar generadores que producen valores aleatorios, es posible generar dos veces un `asset` con el mismo valor para el atributo `assetId` y que ambos se intenten crear en el sistema sin que ninguno se elimine entre ambas llamadas a `create`. Sin embargo, esto puede ser relativamente poco probable dependiendo de la complejidad del identificador. Por ello, para asegurarse de que este caso realmente se comprueba, es posible usar los datos almacenados en el estado, en este caso, es posible usar un `asset` ya creado en el sistema (y, por

tanto, almacenado en el estado en el campo `assets`) como argumento de la operación `create`. De esta forma, es posible comprobar situaciones que con generadores completamente aleatorios se podrían producir con muy poca probabilidad. En este caso concreto, la llamada a la operación `create` en la función `command` sería la siguiente:

```
[{call, ?MODULE, create,
   [eqc_gen:oneof([gen_asset()] ++ S#state.assets)]}]
```

De esta forma, se elige con el generador `oneof`, proporcionado por QuickCheck, o bien un contenido del estado (obtenido del campo `S#state.assets`) siempre y cuando dicha lista no esté vacía, o bien un nuevo `asset` (generado con el generador `gen_asset`). Así, se comprobará el comportamiento de la operación tanto cuando el `asset` no existe (y se espera que se cree satisfactoriamente), como cuando el `asset` ya existe (y se espera un error `duplicated_asset`). Esta misma estrategia se puede repetir para el resto de las operaciones. Por ejemplo, es interesante buscar (operación `find_by_id`), actualizar (operación `update`) o eliminar (operación `delete`) tanto un `asset` que no existe porque no ha sido previamente creado, como uno que sí lo ha sido.

Una vez definidas las operaciones que formarán parte de la especificación de pruebas y el estado interno con sus valores iniciales, es hora de definir las precondiciones y postcondiciones para cada operación, es decir, las restricciones c'_i asociadas a cada una de las operaciones op_i.

En este caso de estudio, cualquier operación puede ser invocada en cualquier momento, sea cual sea el estado del componente, puesto que esto es lo que realmente puede suceder en la realidad. Por tanto, las precondiciones siempre devolverán `true` para todas las operaciones (`C`), sin importar qué información está actualmente almacenada en el estado interno (`S`):

```
precondition(_S, _C) -> true.
```

Con respecto a las postcondiciones, éstas comprueban la ejecución de las operaciones a probar (comprobando su valor de retorno), así como la coherencia del estado interno resultante. Por ejemplo, la operación `create` comprueba, aparte de una serie de condiciones comunes genéricas a otras operaciones definidas en la función `common_postcondition` (como es la comprobación de errores de conexión), si el `asset` fue realmente creado si no existía anteriormente:

```
postcondition(State, {call, ?MODULE, create, [Asset]}, Result) ->
 CreatePostcondition = fun() ->
  case member(Asset, State#state.assets) of
   true -> Result == {error, duplicated_asset};
   false -> Result == ok
  end
 end,
 common_postcondition(State, Result, CreatePostcondition);
```

donde se usa la función anónima (`fun()->... end`), que define el comportamiento específico de la operación `create`, es decir, cuando el `asset` a crear ya existe en el estado (`State#state.assets`) el resultado de la operación debe ser el error `duplicated_asset`, mientras que si éste no existe en el estado y, por tanto, no ha sido creado en el sistema, el resultado de la operación `create` es `ok`. En cualquier caso, el resultado de esta comprobación se almacena en la variable `CreatePostcondition` para ser evaluada posteriormente junto con el resto de comprobaciones genéricas en la función `common_postcondition`.

Finalmente, para que el estado contenga la información adecuada tanto para la generación de datos como para la comprobación de postcondiciones, éste debe ser actualizado después de la ejecución de cada operación. Para esto, es necesaria la definición de la función `next_state`. En este ejemplo, la estructura de la implementación de la función `next_state` sigue el mismo esquema que la función `postcondition`, es decir, se usa una función `common_next_state` con comprobaciones genéricas, y se añade a mayores el comportamiento específico para cada operación concreta. Por ejemplo, esta es la implementación de la función `next_state` para la operación `create`:

```
next_state(State, Result, {call, ?MODULE, create, [Asset]})->
 CreateNextState = fun() ->
  case member(Asset, State#state.assets) of
   false -> add(Asset, State#state.assets);
   true -> State;
  end
 end,
 common_next_state(State, CreateNextState);
```

donde el estado se modifica únicamente cuando no existen errores de conexión (comprobados con la función `common_next_state`), y el `asset` a crear no es uno de los `assets` ya existentes en el estado (`State#state.assets`), puesto que, en otro caso, la operación devolverá un error, en concreto, `duplicated_asset`, de la misma forma que se explicó en la postcondición.

También es importante recalcar que en algunos casos, como ocurre en las operaciones de búsqueda, como `find_by_id` o `find_all`, es decir, operaciones de

lectura o sin efectos colaterales de relevancia para las pruebas, el estado interno no se modificará.

De esta forma, con la definición del estado, la inicialización del mismo, las precondiciones, postcondiciones y las funciones que indican cómo evoluciona el estado con la ejecución de cada operación, se dispone de una especificación completa de pruebas para la API de integración de VoDKA Asset Manager.

6.4.2.3. Validación de la especificación de pruebas

En este caso de estudio, la validación de la especificación de pruebas se ha realizado a través de la implementación de una versión ficticia de VoDKA Asset Manager. Este componente ficticio proporciona la misma API de integración que el componente real, pero simulando su comportamiento, por ejemplo, mantiene la lista de `assets` en un estado interno, sin usar ningún tipo de almacenamiento persistente. De esta forma, es posible ofrecer la misma API de integración que el componente original desde un punto de vista externo, pero sin implementar toda la lógica requerida para cumplir con toda la funcionalidad necesaria para el componente.

La implementación de este componente es muy sencilla, en concreto, es un módulo Erlang de 210 LOC, que ayuda a validar la especificación de pruebas escrita como una máquina de estados QuickCheck.

6.4.2.4. Implementación del componente

En este caso de estudio, este paso consiste en la implementación, y pruebas de unidad, de tres versiones diferentes de la misma API de integración para diferentes tecnologías, en concreto, un servicio web HTTP/XML (HTTPXML), una librería Java (JavaBali) y una aplicación Erlang (ErlBali). De esta forma, una vez terminada esta tarea, será posible disponer de tres implementaciones diferentes de una misma API de integración, en concreto, de la API de integración que cumple los requisitos iniciales establecidos.

El motivo, en este caso, para querer implementar la misma API de integración con tres tecnologías diferentes es facilitar la tarea de integración de otros componentes con el propio componente VoDKA Asset Manager, puesto que, de esta forma, éstos pueden elegir si utilizar una librería Java, una aplicación Erlang o un servicio web HTTP/XML.

6.4.2.5. Implementación de los adaptadores específicos

Además de la especificación de pruebas, escrita a través de un modelo de máquina de estados QuickCheck, y el propio sistema a probar, es decir, el componente VoDKA Asset Manager con sus tres interfaces de acceso, son necesarios otros dos componentes en la arquitectura propuesta (ver figura 6.8):

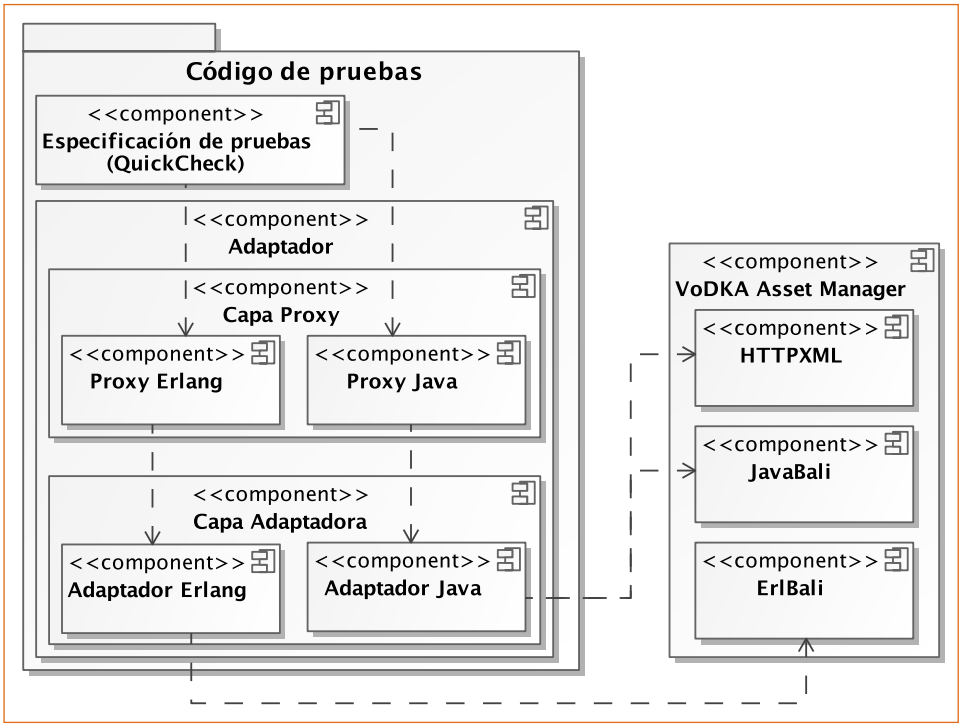


FIGURA 6.8: *Arquitectura propuesta para probar las diferentes implementaciones de la API de integración proporcionada por VoDKA Asset Manager*

- *Capa adaptadora*: en este caso se han creado dos adaptadores, uno para conectar los casos de prueba abstractos con la implementación de la API de integración en Erlang (adaptador Erlang), y otro, basado en Jinterface [38], para conectar estos casos de prueba con la API Java y la interfaz HTTP/XML (adaptador Java).
- *Capa proxy*: esta capa es útil cuando la especificación de pruebas y el adaptador están escritas en lenguajes de programación diferentes, como ocurre para el adaptador Java (puesto que la especificación de pruebas está escrita en Erlang), y se usa para conectar estas dos partes.

El adaptador Erlang

El adaptador Erlang se encarga de conectar los casos de prueba abstractos con la implementación en Erlang de la API de integración. Por tanto, este adaptador proporciona una implementación para que las operaciones abstractas (es decir, las funciones envoltorio op_i') de la especificación QuickCheck invoquen las operaciones de ErlBali.

Para ello, el primer paso es transformar los argumentos usados en las operaciones abstractas, que son enviados al componente real, en el formato esperado por dicho componente. Así, para una operación especificada de forma abstracta como

$op_i(p_i^1, p_i^2, \ldots, p_i^p)$, cuya implementación concreta, rop_i, recibe rp_i parámetros (rp_i^1, rp_i^2, ..., rp_i^{rp}), se propone implementar funciones que transformen los valores abstractos recibidos en dichos parámetros a valores concretos de la siguiente forma:

$$to\_real(p_i^1, \ldots, p_i^p) \rightarrow \{rp_i^1, \ldots, rp_i^{rp}\}$$

A menudo, el número de parámetros de op_i y rop_i son el mismo, es decir, $p = rp$, y la función `to_real`, puede ser definida para cada parámetro como:

$$to\_real(p_i^j) \rightarrow rp_i^j$$

la cual transforma el valor abstracto p_i^j a un valor real rp_i^j, el cual es entendido por el sistema a probar.

Por ejemplo, para la operación `create` es necesario transformar la entidad `asset` que recibe como parámetro dicha operación en el formato abstracto usado en la especificación de pruebas, es decir, una tupla que contiene una lista de propiedades como segundo elemento, al formato esperado por ErlBali, que es un registro del tipo `bali_asset`:

```
to_real_asset(Asset) ->
 #bali_asset {
  id = get_field(Asset, assetId),
  title = get_field(Asset, title),
  summary = get_field(Asset, summary),
  creation_date = get_field(Asset, creationDate),
  run_time = get_field(Asset, runtime)
 }.
```

donde `get_field` es una función que devuelve el valor v_i asociado a la clave k_i en una tupla cuyo segundo elemento es una lista de propiedades, específicamente, $\{Y, [\{k_1, v_1\}, \{k_2, v_2\}, ...]\}$.

Por otro lado, además de adaptar los argumentos de entrada, es necesario procesar las respuestas del sistema a probar, en este caso, ErlBali, para transformarlas en respuestas abstractas entendibles por la especificación de pruebas. Para esto, siguiendo el mismo esquema que para los parámetros de entrada, es posible implementar funciones del estilo:

$$to\_abstract(rr_i \oplus re_i) \rightarrow r_i \oplus e_i$$

que transforman las respuestas reales rr_i o errores re_i obtenidos del componente real en el formato abstracto esperado por el módulo de pruebas QuickCheck, esto es, r_i y e_i.

En el caso de VoDKA Asset Manager, el conjunto de posibles valores devueltos por ErlBali incluyen, entre otros: `ok`, `{error,econnrefused}`, un registro de tipo `bali_asset` o una lista de registros `bali_asset`, los cuales deben ser transformados en valores abstractos usados en la especificación de pruebas:

```
...
to_abstract_response(ok)->
 ok;
to_abstract_response({error, econnrefused})->
 {error, connection_error};
to_abstract_response({ok, Assets}) when is_list(Assets)->
 to_abstract_response(Assets);
to_abstract_response({ok, Asset}) when is_tuple(Asset) ->
 to_abstract_response(Asset);
to_abstract_response([])->
 [];
to_abstract_response(Assets) when is_list(Assets)->
 lists:map(fun(Asset)-> to_abstract(Asset) end, Assets);
to_abstract_response(Asset) when is_tuple(Asset) ->
 {asset, [{assetId, Asset#bali_asset.id},
  {title, Asset#bali_asset.title},
  {summary, Asset#bali_asset.summary},
  {creationDate, Asset#bali_asset.creation_date},
  {runtime, Asset#bali_asset.run_time}]};
...
```

Finalmente, es necesario implementar las funciones adaptadoras aop_i:

$$aop_i(p_i^1, p_i^2, \ldots, p_i^p) \rightarrow r_i \oplus e_i$$

las cuales deben invocar las operaciones reales a probar rop_i, transformando los valores abstractos recibidos como parámetros p_i en valores concretos rp_i, entendibles por el sistema a probar, usando las funciones to_real definidas. De la misma manera, estas funciones también deben transformar, usando las funciones $to\_abstract$ definidas, los valores devueltos $rr_i \oplus re_i$ en respuestas abstractas $r_i \oplus e_i$ para que sea posible comprobar, en la especificación de pruebas, si la operación ha sido ejecutada de la manera esperada.

Así, para cada operación rop_i a probar, la función adaptadora aop_i se escribiría de la siguiente forma:

$$to\_abstract(rop_i(to\_real(p_i^1, \ldots, p_i^p)))$$

la cual, en muchos casos, cuando $p = rp$, puede ser escrita así:

$$to\_abstract(rop_i(to\_real(p_i^1), \ldots, to\_real(p_i^p))))$$

Por ejemplo, la operación `create` se adapta con la siguiente implementación:

```
create(Asset)->
   to_abstract_response(
      bali_server:create_assets([to_real_asset(Asset)])).
```

En este ejemplo, tanto la especificación de pruebas como el adaptador están escritos en el mismo lenguaje de programación (Erlang), por lo que no son necesarias capas adicionales. Por tanto, el *proxy* Erlang que se muestra en la figura 6.8 no tiene que realizar ningún procesamiento adicional aparte de actuar como un *proxy* directo al adaptador, sin manipular los datos.

El adaptador Java

Como se muestra en la figura 6.8, el adaptador Java se usa para invocar las operaciones implementadas en los componentes JavaBali y HTTPXML, es decir, la implementación Java y el servicio web genérico HTTP/XML respectivamente. Aunque en este caso de estudio se usa el adaptador Java para conectar los casos de prueba abstractos con el servicio web HTTP/XML, también podría usarse el adaptador Erlang descrito anteriormente. Sin embargo, se usa esta alternativa para ilustrar con varios ejemplos como probar componentes heterogéneos usando la versión Erlang de QuickCheck independientemente de la tecnología específica con la que el sistema a probar esté construido.

A diferencia del adaptador Erlang y ErlBali, aquí los dos sistemas a probar, JavaBali y HTTPXML, están escritos en un lenguaje de programación diferente del lenguaje de especificación de pruebas, es decir, Erlang. Por tanto, la arquitectura del adaptador Java es más compleja que la del adaptador Erlang. En este caso se ha usado Jinterface [38], una librería Java que permite integrar programas escritos en Java con código Erlang. Además, se toma como base la arquitectura especificada por TTCN-3 para definir la arquitectura del adaptador. Evidentemente, se podría haber usado la arquitectura propuesta por TTCN-3 para construir el adaptador Erlang, pero esto incrementaría la complejidad del marco de trabajo de pruebas innecesariamente cuando la especificación de pruebas y el propio sistema a probar están escritos en el mismo lenguaje de programación.

Siguiendo la estructura de TTCN-3, el adaptador Java se divide en dos partes (ver figura 6.1 de la página 115): una capa de codificación y una capa de adaptación de sistema. Teniendo en cuenta esto, se obtiene una estructura como la mostrada en la figura 6.9, en la que aparecen los paquetes y clases más representativas:

- `codecadapter` (que ha sido implementado utilizando el patrón de diseño *comando* [193]), que, por un lado, codifica las llamadas que proceden de la especificación de pruebas en Erlang, generando objetos de la clase `AdapterRequest`, los cuales representan una invocación de una operación en el sistema a probar; y, por otro lado, envían las respuestas procedentes del sistema a probar a la especificación de pruebas en Erlang, creando para ello objetos de la clase `AdapterResponse`.

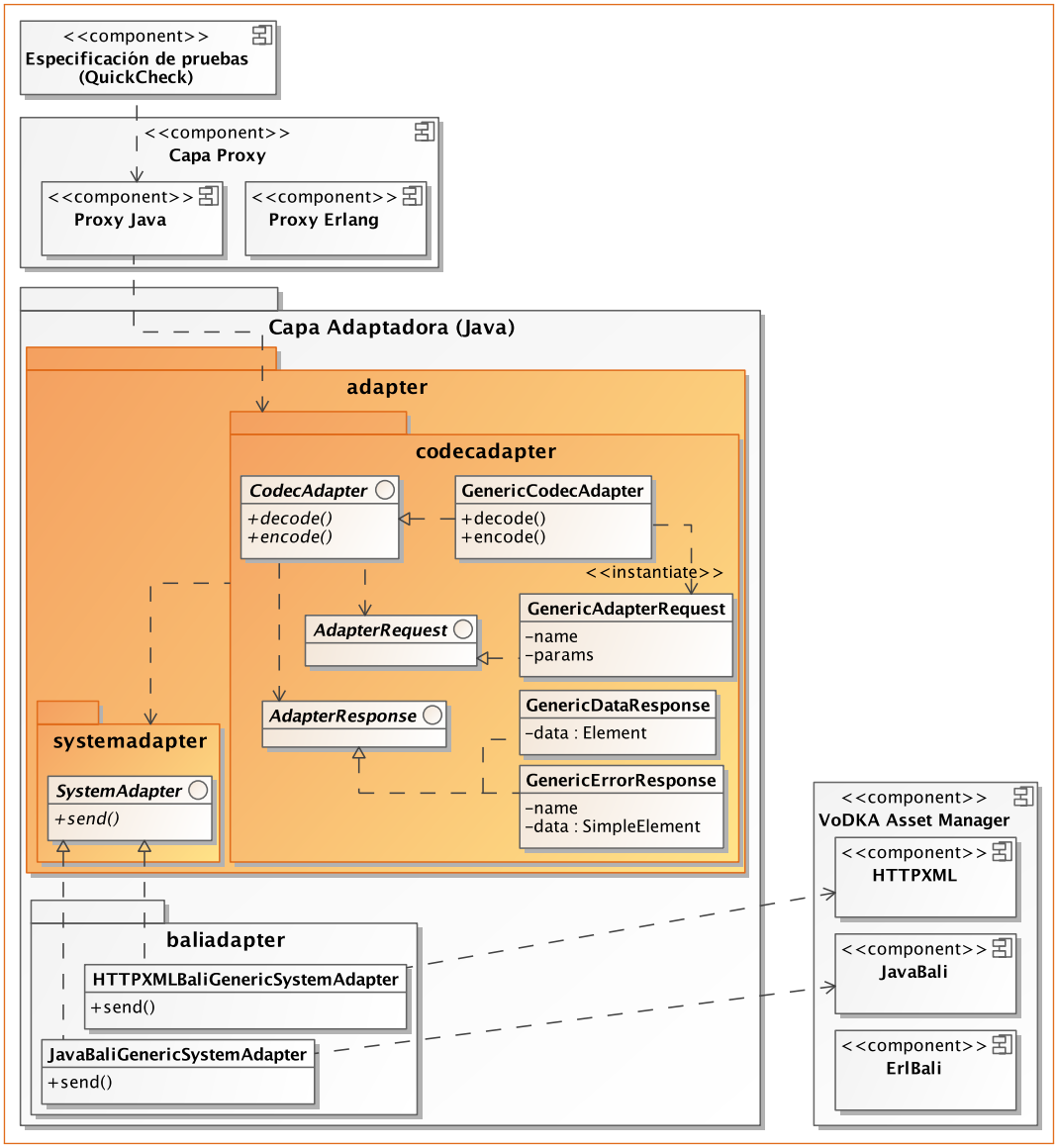


FIGURA 6.9: *Arquitectura (reusable) del adaptador Java*

- `systemadapter`, que es la parte del adaptador que realmente invoca las operaciones del sistema a probar. Estas operaciones están especificadas a través de los objetos de la clase `AdapterRequest` creados por el `codecadapter`.

Mantener la diferencia entre `codecadapter` y `systemadapter` como dos partes diferentes en la estructura del adaptador hace posible la implementación de un componente `codecadapter` genérico que pueda ser usado para probar varias implementaciones de la API de integración, en concreto, JavaBali y HTTPXML, usando sólo implementaciones específicas del componente `systemadapter`.

Para ello, es necesario definir una codificación genérica de los mensajes intercambiados con el sistema a probar. Así, la invocación de la operación rop_i con los parámetros $rp_i^1, rp_i^2, \ldots, rp_i^{rp}$, se codifica como una tupla:

$$\{rop_i, [erp_i^1, erp_i^2, \ldots, erp_i^{rp}]\}$$

donde erp_i^j es también una tupla:

$$\{\{type, rpd_i^j\}, \{value, rpv_i^j\}\}$$

en la que rpd_i^j es el tipo de dato del parámetro rp_i^j (es decir, `string`, `integer`, `list`, ...), y rpv_i^j es el valor que toma el parámetro rp_i^j. Por ejemplo, la operación `find_by_id(<ID>)`, que recupera los datos asociados al `asset` con identificador `<ID>` (o devuelve un error si éste no existe), se codifica como:

```
{find_by_id, [{{type, string}, {value, <ID>}}]}.
```

Esta transformación la realiza la capa *proxy*, la cual se debe implementar en el mismo lenguaje de programación usado para definir la especificación de pruebas, es decir, Erlang. A diferencia del adaptador Erlang, aquí la capa *proxy* sí es necesaria para enviar los datos al adaptador, el cual está implementado en Java. Por ejemplo, la función `find_by_id` se implementa en la capa *proxy* de la siguiente manera:

```
find_by_id(AssetId) ->
 proxy_java:call(find_by_id,
    [{{type, string}, {value, AssetId}}]).
```

donde `proxy_java` es un módulo genérico que envía mensajes al adaptador Java y recibe las respuestas enviadas por dicho adaptador, usando para ello la librería Jinterface. Por ejemplo, esta es la implementación de la función `call` del módulo `proxy_java`:

```
call(Operation, Params)->
 {testservice, 'testnode@localhost'} !
  {self(), {Operation, Params}},
 receive
  Any ->
   Any
  after 5000 ->
   throw({error, timeout})
  end.
```

donde `testnode@localhost` es el nombre del nodo Erlang instanciado por la librería Jinterface, y `testservice` el nombre del "buzón" empleado para recibir mensajes en el adaptador:

```
...
OtpNode self = new OtpNode("testnode@localhost");
OtpMbox mbox = self.createMbox("testservice");
...
```

Cuando los mensajes enviados por el *proxy* llegan a la capa adaptadora, estos son codificados usando el método `encode` implementado en la clase de codificación `GenericCodecAdapter`, generando así objetos pertenecientes a la clase `GenericAdapterRequest`, que representan una invocación a una operación. Estos objetos son procesados por las clases correspondientes del `systemadapter` para invocar las operaciones correspondientes del sistema a probar representadas por estos objetos.

En contraste con el `codecadapter`, cuya implementación es la misma para los diferentes sistemas, es necesario especificar un `systemadapter` diferente para cada tecnología a probar. En este caso de estudio se han implementado dos `systemadapter` específicos: `HTTPXMLBaliGenericSystemAdapter` para probar el componente HTTPXML y `JavaBaliGenericSystemAdapter` para JavaBali.

Por ejemplo, cuando el adaptador recibe un mensaje de creación de un `asset`, es decir, cuando se invoca la operación `create` con los datos del `asset` a crear, la clase `HTTPXMLBaliGenericSystemAdapter` debe realizar una petición POST HTTP a una URL específica, en concreto, `/adi/create_assets.yaws`, incluyendo en el cuerpo de dicha petición los datos del `asset` a ser creado en formato XML, como se muestra a continuación:

```
...
if (baliHTTPComm != null) {
 Asset asset = toObject(
    genericAdapterRequest.getParams().get(0));
 InputStream inputStream = baliHTTPComm.doPostInputStream(
    "/adi/create_assets.yaws", toObject(asset));
 HTTPXMLResponse response =
    HTTPXMLBaliSystemCodec.getResponse(inputStream);
 testCommunication.enqueueMsg(from, response, false);
} else {
 testCommunication.enqueueMsg(from,
    new GenericErrorResponse("not_started", null));
}
...
```

En este fragmento de código se observa como, si el sistema ha sido inicializado, se obtiene el objeto `Asset` de los parámetros de la operación a través del objeto `genericAdapterRequest`, se realiza una petición POST usando el método

`doPostInputStream` y, finalmente, se obtiene el resultado para ser enviado a la capa Erlang a través del método `enqueueMsg`. Por otra parte, si el sistema no ha sido inicializado, la especificación de pruebas recibirá el error `not_started`.

Por otro lado, la implementación de `JavaBaliGenericSystemAdapter` es diferente, puesto que no debe invocar una URL, sino que debe usar el componente JavaBali para invocar las operaciones correspondientes. Para ello, instancia una fachada, llamada `AssetFacade`, definida por la librería JavaBali, e invoca el método que corresponda, en este caso, `addAsset`:

```
...
if (assetFacade != null) {
 try {
  Asset asset = toObject(
     genericAdapterRequest.getParams().get(0));
  assetFacade.addAsset(asset);
  testCommunication.enqueueMsg(from,
     new GenericDataResponse(
        new SimpleElement("ok", ElementType.ATOM)));
 } catch(DuplicateInstanceException die) {
  testCommunication.enqueueMsg(from,
     new GenericErrorResponse("duplicated_asset", null));
 }
} else {
 testCommunication.enqueueMsg(from,
    new GenericErrorResponse("not_started", null));
}
...
```

Una vez que el adaptador invoca la operación real en el sistema a probar, y recibe la respuesta asociada a dicha invocación, esta respuesta se convierte en un objeto `AdapterResponse`. De la misma forma que cada operación tiene asociada una respuesta y sus posibles errores (como se muestra en la tabla 6.1 de la página 140), este objeto puede ser una instancia de la clase `GenericDataResponse`, si la operación se ejecuta satisfactoriamente, o de la clase `GenericErrorResponse`, si la operación devuelve un error.

En cualquier caso, el mensaje se encola para poder ser procesado por el sistema `codecadapter` de nuevo (específicamente, por la clase `GenericCodecAdapter`), el cual decodifica el mensaje y lo envía de vuelta a la especificación de pruebas Erlang. Para esto, el método `decode` crea los objetos Jinterface correspondientes que representan la respuesta real del sistema a probar en la forma abstracta definida, es decir, una tupla cuyo segundo elemento es una lista de propiedades.

Esta arquitectura permite reusar diferentes componentes, los cuales pueden ser distribuidos como un fichero *jar* en forma de librería de pruebas para poder ser

usados en otros proyectos, evitando, de esta forma, crear adaptadores Java desde cero una y otra vez. Así, además de la estructura genérica para los adaptadores, es posible reusar algunas partes específicas del adaptador como es el `codecadapter`, y algunas utilidades comunes del paquete `adapter` mostrado en la figura 6.9, en concreto, aquellos paquetes y clases que se muestran destacadas en esta figura.

Por ejemplo, usando esta librería genérica de pruebas para probar la especificación de VoDKA Asset Manager, aparte de escribir la especificación de pruebas e implementar las transformaciones que realizará la capa *proxy*, simplemente es necesario implementar los adaptadores específicos de sistema, esto es, el paquete `baliadapter` mostrado en la figura 6.9. Este paquete contiene las clase `HTTPXMLBaliGenericSystemAdapter` para invocar las operaciones de la interfaz HTTPXML, y `JavaBaliGenericSystemAdapter` para invocar las operaciones del componente JavaBali. El resto de clases son reutilizables, y pueden usarse en otros proyectos.

6.4.2.6. Generación y ejecución de los casos de prueba

La ejecución de las pruebas con este caso de estudio produce la generación de llamadas a la API de integración de VoDKA Asset Manager, generando así secuencias aleatorias de llamadas con las operaciones especificadas en la función `command`. Por ejemplo, esta es una secuencia aleatoria de operaciones generada por QuickCheck para este caso de estudio:

```
delete("A")
init("http://localhost:8888","user","password")
find_by_id("A")
create({asset, [{assetId,"A"}, {creationDate,"59958230400"},
   {title,"A"}, {summary,"A"}, {runtime,"0:00:00"}]})
find_by_id("A")
find_by_id("B")
...
```

donde tanto la secuencia de operaciones, como los argumentos que recibe cada operación son producidos por QuickCheck a través de los generadores, como son `gen_asset` o `gen_valid_id`.

Por otro lado, es importante destacar que esta misma especificación abstracta es válida para probar todas las implementaciones de la API de integración. Para ello, únicamente hay que cambiar el adaptador usado, sin necesidad de implementar de nuevo la especificación de pruebas.

6.4.2.7. Análisis de los resultados de las pruebas

La ejecución de las pruebas usando esta aproximación reveló varios errores en las diferentes implementaciones de la API de integración de VoDKA Asset Manager.

En concreto, mientras que la API HTTPXML cumple la especificación de la API de integración, existen algunas propiedades que tanto las implementaciones ErlBali como JavaBali no cumplen.

Así, por ejemplo, se detectó que la implementación JavaBali no devolvía un error `not_found` cuando la operación `update` se aplicaba sobre un `asset` no existente. Por su parte, ErlBali contenía defectos similares, en concreto, en las operaciones `update` y `create_asset`, las cuales no retornaban los errores esperados en algunas situaciones, así como la operación `delete`, la cual no funcionaba en absoluto puesto que no realizaba el borrado correctamente.

Estos errores fueron detectados por QuickCheck, mostrando tanto la traza completa que causó el error, así como una traza minimizada que causa el mismo error obtenida a partir del proceso de reducción aplicado por QuickCheck. Por ejemplo, la traza minimizada mostrada por QuickCheck cuando encuentra que la operación `update` no devuelve el error `not_found` cuando se intenta actualizar un `asset` no existente es la siguiente:

```
Shrinking..........(10 times)
[{set,{var,3},
    {call,test_asset_eqc,init,[
      "http://localhost:8888","user",
      "password"]}},
 {set,{var,4},
    {call,test_asset_eqc,update,
        [{asset,
           [{assetId,"A"},
            {creationDate,"59958230400"},
            {title,"A"},
            {summary,"A"},
            {runtime,"0:00:00"}]}]}}]
```

En este caso, el número mínimo de pasos para obtener este error consiste en autenticarse en el sistema usando la operación `init` con un usuario y contraseña válidos, y, posteriormente, sin haber creado ningún `asset` en el sistema, llamar a la operación `update`, la cual recibe como argumento un `asset` no existente (puesto que no se ha creado ninguno anteriormente).

En resumen, gracias a esta aproximación, se han conseguido detectar diferentes inconsistencias entre las diferentes implementaciones de la API de integración y la especificación de la misma. Además, el hecho de que estas implementaciones fueran probadas por sus desarrolladores anteriormente convierte este hecho en algo sorprendente, no porque ahora se hayan encontrado defectos que no hayan aparecido anteriormente, sino porque éstos afecten a operaciones básicas de un sistema real que está siendo usado en despliegues reales. Probablemente, la explicación de

este hecho es que las diferentes implementaciones de VoDKA Asset Manager no se usan de la misma manera, puesto que no todas las operaciones se usan en todas las implementaciones. Por tanto, las operaciones con defectos en su implementación revelan que dichas funcionalidades no han sido usadas hasta ahora en las correspondientes implementaciones erróneas de la API de integración. En cualquier caso, se han localizado y solucionado defectos que podrían haber sido identificados antes de la etapa de despliegue y, de esta forma, se evitan situaciones críticas en futuros despliegues en producción.

6.4.3. Resultados de aplicar la metodología de pruebas

El primer punto que es importante destacar es que el uso de esta metodología permitió detectar defectos en componentes que están actualmente en uso en despliegues reales. Para ello, solamente ha sido necesaria una única especificación de pruebas abstracta, independiente de la estructura y lenguaje de programación del sistema a probar. Esta especificación permitió probar tres implementaciones diferentes de la misma especificación funcional de la API de integración, evitando interpretaciones diferentes de los requisitos a probar en cada uno de los componentes.

Con una aproximación más tradicional, no basada en la estructura de TTCN-3, es muy probable que se creasen tres conjuntos de pruebas para probar las tres implementaciones de la API de integración, probablemente cada uno de ellos implementado en un lenguaje diferente, dependiente del lenguaje del propio sistema a probar. Por otra parte, una de las implementaciones podría haber servido como referencia para probar las otras [260], aunque, en cualquier caso, con un mayor esfuerzo en las actividades de pruebas.

El caso de la API de integración de VoDKA Asset Manager no es algo único. El uso de APIs de integración es un mecanismo estándar para integrar componentes, por lo que el diseño de una aproximación estructurada y repetible para probar este tipo de implementaciones de APIs de integración es útil, no sólo para este caso de estudio, sino, en general, para otro tipo de componentes.

A continuación se detallan algunos aspectos clave en el uso de esta metodología, los cuales se extraen como consecuencia de su aplicación al caso de estudio.

6.4.3.1. Cantidad de código de pruebas

A pesar de que esta aproximación requiere una infraestructura compleja (por el uso de capas y componentes intermedios), la arquitectura de pruebas propuesta representa una reducción considerable de código fuente necesario para producir los casos de prueba (ver figura 6.10). Usando líneas de código como medida de referencia, éstos son los tamaños de cada uno de los componentes involucrados en la arquitectura de pruebas:

- Especificación de pruebas: 473 LOC.

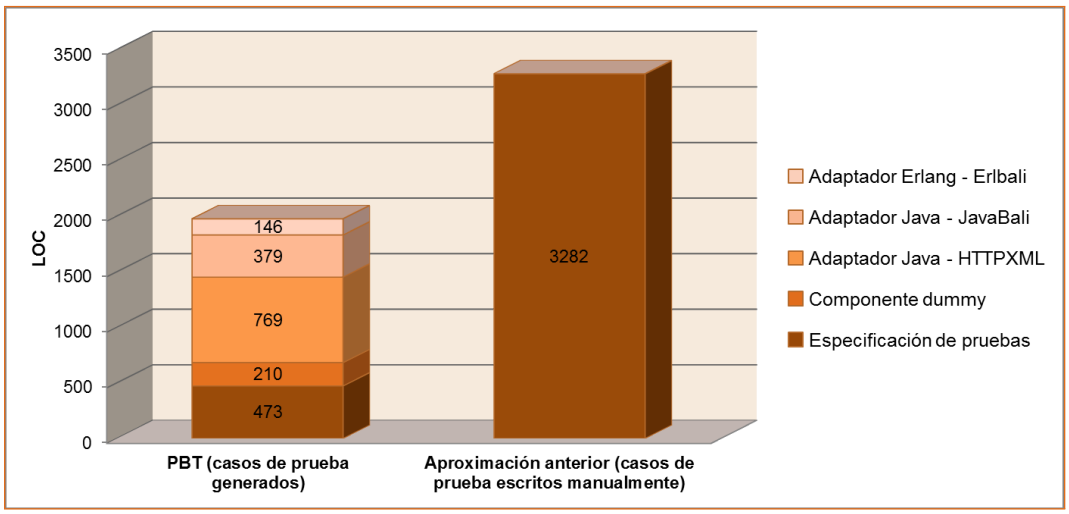


FIGURA 6.10: *Comparación de líneas de código entre aproximación de pruebas basada en propiedades abstractas y aproximación con casos de prueba escritos manualmente*

- Adaptadores: ∼1850 LOC.
  - Adaptador Erlang: ∼150 LOC.
  - Adaptador Java: ∼1700 LOC, donde:
    - el adaptador para JavaBali tiene ∼400 LOC,
    - el adaptador para HTTPXML ∼750 LOC,
    - y el código restante (un 30 % de las 1700 LOC) son componentes reutilizables para otros proyectos.
- Componente ficticio: 210 LOC.
- SUTs: 15,5K LOC.

En total, sin tener en cuenta el código reusable, se han necesitado unas 2K líneas de código de pruebas específico para probar las tres implementaciones de la API de integración de VoDKA Asset Manager. El código de pruebas ya existente, anterior a este estudio, para probar las tres implementaciones de la API de integración de VoDKA Asset Manager, obtenido con una aproximación en la que los casos de pruebas se escriben manualmente (usando JUnit para probar la implementación Java, JUnit y HttpUnit para probar el servicio web HTTPXML, y Eunit para probar la implementación Erlang), tenía alrededor de 3,3K LOC. Esto, por tanto, representa una reducción del 39 % de LOC cuando se usa la aproximación propuesta, a la par que ahora hay partes reutilizables y con una aproximación más eficaz, puesto que se han encontrado defectos ocultos.

Además, aunque las operaciones probadas son las mismas en ambas aproximaciones, y los casos de prueba son similares (crear un contenido, posteriormente recuperarlo y comprobar si se ha creado satisfactoriamente; actualizar un contenido ya existente y comprobar si fue realmente actualizado; eliminar un contenido y comprobar que éste ya no existe en el sistema, etc.), la aproximación basada en propiedades genera muchos más casos de prueba que los casos escritos con la aproximación manual, incluyendo muchas más combinaciones de secuencias de llamadas a operaciones. Así, mientras que en la aproximación manual se comprueba que dado un contenido, con datos concretos escritos manualmente, se crea, se recupera, actualiza y finalmente se borra, en la aproximación basada en propiedades únicamente se especifican las operaciones disponibles (`create`, `find_by_id`, `find_all`, `update` o `delete`), y es la propia herramienta de pruebas basadas en propiedades, en este caso, QuickCheck, la que genera secuencias aleatorias de comandos que ejecutan estas operaciones, usando múltiples contenidos con datos aleatorios generados como argumentos de entrada de estas operaciones.

Por tanto, con un 39 % menos de código de pruebas, la aproximación basada en propiedades permite probar la especificación con más casos de prueba, los cuales, además de los casos de prueba que han sido escritos manualmente, contienen muchas otras combinaciones de operaciones y datos de entrada, los cuales no han sido usados en la aproximación manual existente hasta la realización de este estudio.

En general, esta propuesta requiere que los probadores de software sigan la arquitectura presentada, en la que el código de pruebas se divide en una especificación de pruebas y un adaptador. Sin embargo, usar esta aproximación permite probar un sistema desde un punto de vista de más alto nivel, definiendo propiedades en vez de casos de prueba específicos, resultando en una aproximación más exhaustiva y objetiva con menos código fuente de pruebas, y evitando la reimplementación de la especificación de pruebas para cada implementación del sistema a probar. Por tanto, puesto que el esfuerzo requerido es menor para producir los casos de prueba, esta es una aproximación más eficiente, y así, es posible, o bien reducir los costes de realizar las pruebas, o bien generar más casos de prueba dedicando más tiempo a la ejecución automatizada de secuencias de pruebas a partir de la especificación de las mismas, incrementando la confianza en el software, con el mismo coste.

6.4.3.2. Mantenimiento y reusabilidad código de pruebas

Como los requisitos normalmente cambian menos que la implementación, las especificaciones de pruebas basadas en requisitos son más estables y menos sujetas a cambios que las pruebas basadas en casos de prueba escritos manualmente o generadas con aproximaciones que se basan en la implementación del sistema a probar.

Además, en contraste con otras aproximaciones en las que el código de pruebas es, a menudo, un componente indivisible, en esta aproximación el código de pruebas está dividido en dos partes: la especificación de pruebas, la cual depende de

la funcionalidad a probar, y los adaptadores, que dependen de la implementación concreta del sistema a probar. Por un lado, la especificación de pruebas permite tener más control sobre cómo afectan a las pruebas las modificaciones en el ciclo de vida del componente. Por otro lado, la existencia de adaptadores permite reusar una gran parte de la arquitectura de pruebas, puesto que normalmente se tendrá un adaptador por cada uno de los lenguajes de programación candidatos. Así, crear los adaptadores puede ser costoso inicialmente, pero al ser altamente reutilizables entre diferentes proyectos, este coste se compensa.

Por último, el desarrollo de componentes ficticios para validar la implementación de la especificación de pruebas también requiere un esfuerzo adicional, algo que puede ser difícil de argumentar. Evidentemente, existe un riesgo de que la implementación de estos componentes sea muy similar a la implementación del componente real. Sin embargo, es importante destacar que el uso de un componente ficticio en esta metodología se sugiere para que en el flujo de trabajo sea posible paralelizar diferentes tareas, en concreto, la validación de la especificación de pruebas con la implementación del componente, como se muestra en la figura 6.5 de la página 130. La implementación del componente ficticio es algo opcional, puesto que la especificación de pruebas puede ser validada sin usar ningún tipo de componente ficticio (por ejemplo, usando herramientas de *mocking* en la especificación de pruebas), o bien usando el componente real, o una versión inicial (o prototipo) del mismo.

6.4.3.3. El uso de una aproximación basada en propiedades y QuickCheck

QuickCheck, la herramienta elegida para generar y ejecutar los casos de prueba, facilita la definición abstracta de estos casos de prueba sustancialmente. Primeramente, la herramienta QuickCheck usa un lenguaje de propósito general como lenguaje de especificación, en este caso, Erlang, sin obligar a usar ningún lenguaje específico para este fin. En segundo lugar, QuickCheck genera automáticamente casos de prueba a partir de las propiedades formuladas, forzando a la persona que lo utilice a pensar en el componente a ser probado de una forma más abstracta, sin tener en cuenta los detalles de implementación.

Cuando se usa una aproximación basada en propiedades, evidentemente, los probadores de software necesitan conocer cómo escribir propiedades en lugar de casos de prueba específicos. Este hecho supone un desafío en la mayoría los casos, puesto que representa un cambio de mentalidad, sobre todo en el nivel abstracción de las pruebas [277]. Esto puede ser considerado un inconveniente de esta aproximación, pero si se compara con otro tipo de aproximaciones, escribir manualmente un conjunto de pruebas completo, objetivo y exhaustivo tampoco es una tarea trivial [180]. Además, la aproximación propuesta tiene la ventaja de que los probadores de software no tienen que ser conscientes del lenguaje del programación en el que está implementado el sistema a probar, simplemente su especificación funcional, puesto que el lenguaje de especificación de pruebas es siempre Erlang. Por

tanto, los probadores de software podrían convertirse en especialistas competentes en las herramientas específicas de pruebas y el lenguaje de especificación usado en las mismas.

Una preocupación habitual cuando se utilizan herramientas de prueba automáticas que generan aleatoriamente datos de pruebas y/o casos de prueba es la reproducibilidad de las pruebas. Sin embargo, específicamente, dentro de la herramienta QuickCheck, hay un número de funciones de utilidad que permiten tanto reproducir secuencias de casos de prueba que han provocado un error como controlar los casos de prueba que se generan a partir de una especificación dada, con el fin de supervisar la distribución de datos y evaluar la ausencia de sesgo. Así, es posible especificar, como parte de la especificación de la prueba en sí, una distribución de probabilidad (es decir, la frecuencia de cada uno de los comandos) para las operaciones en un modelo de máquina de estados. Además, QuickCheck también permite guardar casos de prueba generados en base a algún criterio (normalmente cobertura) para usarlos como pruebas de regresión.

Por otro lado, las capacidades de reducción de contraejemplos de QuickCheck son de gran ayuda cuando se produce un error, puesto que proporcionan formas sencillas de reproducir los contraejemplos reportados por la herramienta, que hacen que la depuración de errores y su posterior resolución sea considerablemente más fácil [87, 135, 374].

Además, como se ha explicado anteriormente, la generación de casos de prueba a partir de propiedades generalmente produce como resultado casos de prueba que tienen en cuenta muchas más combinaciones de situaciones y, además, con mucho menos código fuente de pruebas necesario que otras aproximaciones en las que los casos de prueba deben ser escritos manualmente. Así, usando esta aproximación, se deben escribir únicamente propiedades abstractas que el software debe satisfacer, y QuickCheck es la herramienta responsable de generar casos de prueba que comprueben estas propiedades.

En el caso concreto de este caso de estudio se ha escrito una especificación de pruebas de 473 LOC que especifica dos terceras partes de las operaciones de la API de VoDKA Asset Manager. Esto representa aproximadamente un total de 9,3K LOC de la implementación del sistema a probar (el cual tiene en total un total de 15,5K LOC). Es importante destacar que, con únicamente 473 LOC, es posible especificar, usando la versión Erlang de QuickCheck, el comportamiento funcional de 9,3K LOC, como se muestra en la figura 6.11.

6.4.3.4. Comparación con TTCN-3

Comparando y contrastando esta aproximación con TTCN-3, de la cual se ha tomado su arquitectura como inspiración, se puede observar cómo en ambas aproximaciones las pruebas se especifican de manera abstracta, derivada de los requisitos

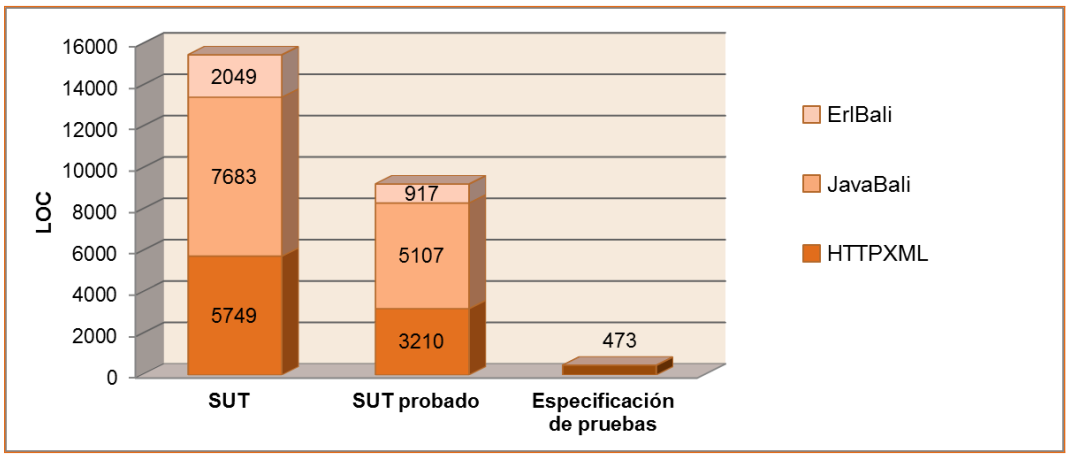


Figura 6.11: *Comparación de líneas de código entre la especificación de pruebas y el SUT*

funcionales. En TTCN-3, el ATS se escribe manualmente a partir de las especificaciones de sistema y requisitos, mientras que en esta aproximación se escriben propiedades abstractas que describen el comportamiento del sistema a probar (a partir de las cuales se genera el ATS automáticamente). Por otro lado, ambas aproximaciones usan un adaptador que se encarga de conectar el conjunto de pruebas abstractas con el sistema a probar, transformando cada llamada a operaciones abstractas en llamadas concretas a operaciones proporcionadas por dicho sistema a probar.

Al igual que en TTCN-3, en esta aproximación las especificaciones de pruebas se escriben siempre en el mismo lenguaje de programación, independientemente del lenguaje de programación en el que el sistema a probar esté implementado. Este hecho representa una ventaja importante sobre otras aproximaciones en las que la especificación de pruebas se escribe en el mismo lenguaje que el sistema a probar, puesto que aquí los probadores de software únicamente necesitan tener habilidades en un lenguaje de especificación, en este caso, el lenguaje de programación Erlang.

Por otro lado, en contraste con Erlang, que es un lenguaje de programación de propósito general, TTCN-3 es un lenguaje estándar específico para la realización pruebas con soporte explícito para muchos conceptos útiles en las tareas de pruebas: veredictos, plantillas de coincidencia de patrones, pruebas concurrentes y no concurrentes, etc. Además, existen entornos de desarrollo integrados (IDEs) para escribir pruebas y adaptadores [52, 68, 70] que convierten la ejecución de las pruebas y la inspección de los resultados en tareas más sencillas, así como herramientas que facilitan la construcción de adaptadores TTCN-3 [62]. Sin embargo, que TTCN-3 no use un lenguaje de propósito general hace que nunca se dé el mejor caso, es decir, la adaptación trivial, que en el caso de estudio de VoDKA Asset Manager se produce a la hora de realizar las pruebas de ErlBali.

La aproximación propuesta, por su parte, tiene las ventajas de usar propiedades para la realización de las pruebas, evitando escribir todos los casos de prueba uno a uno, y obteniendo como resultado una especificación de pruebas más estable, y menos cambiante ante modificaciones en la implementación del sistema a probar. Por otro lado, debido al uso de un lenguaje de programación estándar, como Erlang, para escribir las propiedades de prueba, la versatilidad de esta aproximación es mayor, puesto que permite a la persona que la use utilizar cualquier estructura permitida en el lenguaje de programación. Además, el uso de la versión Erlang de QuickCheck permite usar todas las características que QuickCheck proporciona: generadores, máquinas de estados, búsqueda de contraejemplos reducidos, etc., lo cual ayuda tanto a producir mejores conjuntos de prueba, como a entender mejor los fallos que se producen con los mismos.

6.4.3.5. Adopción y uso de la metodología

La metodología explicada describe los pasos para probar la implementación de una API de integración, así como las personas y roles involucrados en cada fase. Así, los programadores y los probadores de software tienen diferentes roles en esta metodología: mientras que los programadores centran su trabajo en la implementación del sistema a probar, implementando las pruebas de unidad necesarias para comprobar el comportamiento interno esperado, los probadores de software son los responsables de definir la especificación de pruebas que representan el comportamiento de dicho sistema desde un punto de vista externo.

Por un lado, definir una especificación de pruebas es un trabajo manual que se puede realizar basándose en los requisitos del sistema a probar. Los probadores de software deben usar la especificación del sistema a probar para escribir las propiedades que representan el comportamiento del mismo. Por otro lado, esta aproximación requiere la implementación de una capa adaptadora que conecte la especificación de pruebas con el sistema a probar. Aunque es posible argumentar que el adaptador hace que esta aproximación sea más compleja, esta capa es el componente clave que permite el uso de una misma especificación de pruebas para probar diferentes implementaciones de una misma especificación de una API de integración. Idealmente, esta capa debería implementarse con la cooperación de los probadores de software y programadores, puesto que podría ser necesario conocer detalles de bajo nivel sobre cómo invocar las operaciones del sistema a probar.

El esfuerzo de implementar el adaptador depende del lenguaje de programación en el que el sistema a probar esté construido. Así, cuando este sistema está implementado en Erlang, la capa adaptadora es muy simple, puesto que la especificación de pruebas también está escrita en Erlang. Sin embargo, cuando el sistema está implementado en un lenguaje de programación diferente, esta capa es algo más compleja.

Aunque no es posible automatizar la generación de la capa adaptadora de manera genérica, puesto que es necesario conocer cómo se invoca cada operación (realizando una llamada a un método de una clase Java, invocando una URL por POST, etc.), y esta información normalmente no está presente en la descripción del sistema, sí es posible implementar componentes reusables que faciliten la implementación de los adaptadores. Por ejemplo, en el caso de estudio de VoDKA Asset Manager, se han desarrollado tanto clases Java como un protocolo de comunicación común y genérico entre la capa Erlang que define la especificación de pruebas y la capa adaptadora, que permite reusar muchas partes del adaptador Java.

En caso de que el sistema a probar estuviera implementado en otro lenguaje de programación, como, por ejemplo, C, se recomienda seguir la misma aproximación que se ha explicado anteriormente para el adaptador Java, es decir, desarrollar una estrategia común que permita reusar tantas partes del adaptador como sea posible en otros proyectos. El principal objetivo de estas partes reusables es ocultar el protocolo de comunicación entre la capa Erlang y la capa adaptadora (`codecadapter`), y que únicamente sea necesario implementar cómo cada operación concreta es realmente invocada (`systemadapter`).

En resumen, usando esta aproximación de pruebas es posible probar un componente desde una perspectiva de caja negra, es decir, sin tener en cuenta la estructura interna del sistema a probar. Para ello, únicamente se debe definir una especificación abstracta de pruebas (basada en propiedades) y el código correspondiente del adaptador que se encarga de invocar las operaciones reales del sistema a probar, es decir, la capa `systemadapter` en el adaptador Java.

6.5. Generación automática de propiedades abstractas a partir de UML y OCL

Una posible mejora a la metodología de pruebas presentada en la sección 6.4 es aligerar la tarea de definir las propiedades de pruebas QuickCheck. Para ello, la aproximación propuesta es que dichas propiedades se generen automáticamente a partir de un modelo del sistema a probar, de la misma manera que se realiza en las pruebas basadas en modelos explicadas en la sección 6.2.2, evitando, de esta forma, que los probadores de software tengan que escribirlas manualmente.

Así, dentro de los componentes que forman parte de la arquitectura de pruebas que se muestra en la figura 6.6 de la página 135, la especificación de pruebas no tendría que ser escrita manualmente, sino que se generaría automáticamente a partir de un modelo del sistema a probar. El resto del proceso sería exactamente el mismo, es decir, la especificación de pruebas QuickCheck se usa para generar un ATS que podrá ser ejecutado gracias al uso de un adaptador que conecta dichos casos de prueba abstractos con el propio sistema a probar.

A continuación se describirá con detalle la arquitectura de pruebas requerida para generar automáticamente la especificación de pruebas QuickCheck a partir de modelos de especificación, y cómo funciona esta aproximación cuando se prueban componentes sin estado, componentes con estado y componentes parcialmente especificados [187].

6.5.1. Arquitectura de pruebas

La arquitectura de pruebas modificada combina las aproximaciones mostradas en las figuras 6.6 (página 135) y 6.3 (página 120), combinando, por tanto, las *pruebas basadas en propiedades* con las *pruebas basadas en modelos*. Se adquieren así las ventajas de ambas aproximaciones, las cuales se traducen en que es posible probar el software de una forma más eficiente en cuanto a la menor cantidad de código de pruebas y esfuerzo requerido.

La figura 6.12 muestra los componentes que forman parte de dicha arquitectura. Como se observa, este diagrama es una especialización del que se muestra en la figura 6.3 de la página 120, con las siguientes peculiaridades:

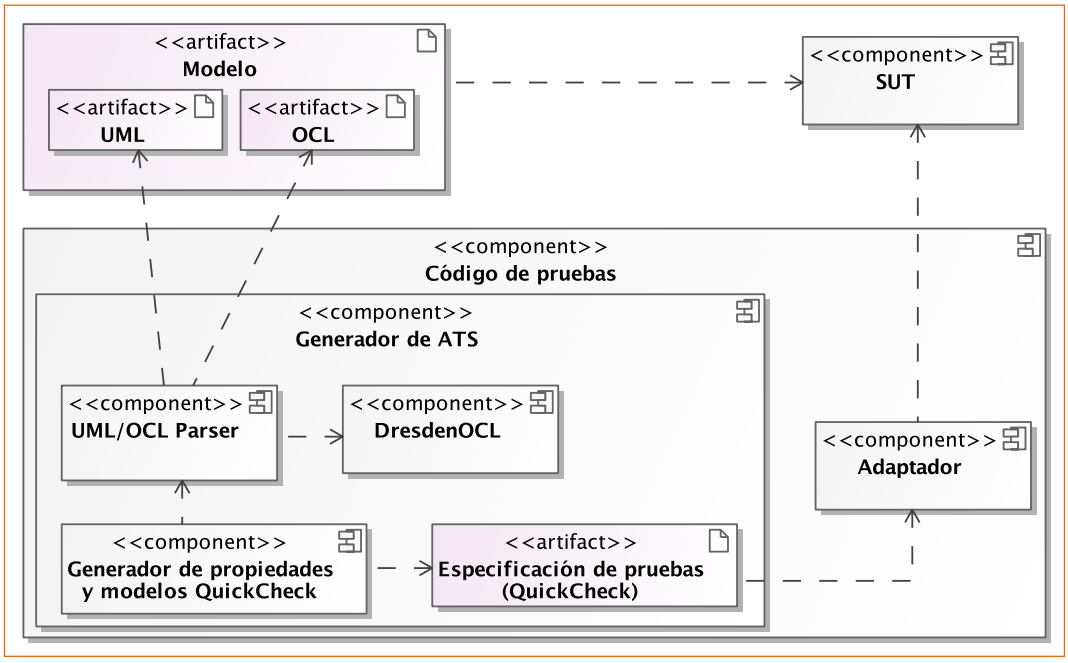


FIGURA 6.12: *Arquitectura propuesta para probar APIs de integración combinando modelos con propiedades*

- Se usan UML y OCL como lenguajes para crear el modelo.
- El ATS se genera con propiedades, que a su vez se generan a partir del modelo especificado con UML y OCL.

- De las diferentes aproximaciones que existen para conectar el ATS con el sistema a probar, se usa un adaptador.

En este caso particular, se usan diagramas de clase UML para modelar el sistema a probar. Estos diagramas representan las clases (o componentes) que forman parte de un sistema, organizadas en paquetes, y mostrando por cada clase sus atributos (con sus tipos de datos), operaciones (con los parámetros de entrada y de salida) y relaciones con otras clases (asociaciones, dependencias de uso, etc.).

La elección del lenguaje UML está motivada por ser un lenguaje de modelado de propósito general, conocido ampliamente tanto en la industria como en el ámbito académico. Además, al contrario que otras alternativas que podrían haber sido usadas como JML [240], iContract [231], jContractor [226], o jContract [53] para Java, o Spec# [99] para C#, que incluso algunas de ellas también proporcionan herramientas que usan las precondiciones y postcondiciones definidas para generar valores de entrada para probar las operaciones, como es el caso de JMLUnit [141] (JML) o JTest [53] (jContract), UML es un lenguaje independiente de la implementación.

Por otra parte, los diagramas UML se pueden complementar con información adicional para expresar información que éstos no pueden capturar. Existen varias alternativas, como es el uso de SQL u otros lenguajes, pero se usará el lenguaje OCL por estar estrechamente integrado con el uso de UML. OCL [129, 363] es un lenguaje formal declarativo usado para definir restricciones sobre objetos adoptado por el grupo OMG como parte de UML 2.0 [281]. En concreto, OCL ofrece mecanismos predefinidos para obtener el valor de un objeto, navegar a través de una relación entre objetos, iterar sobre colecciones de objetos, etc. Así, puede ser usado para describir valores válidos para los atributos, expresar precondiciones y postcondiciones para las operaciones, invariantes que se deben cumplir para una clase, expresar consultas, etc. Para ello, el lenguaje OCL incluye una librería estándar de tipos de datos, la cual define una serie de tipos de datos simples (`Real`, `Integer`, `String`, `Boolean` y `UnlimitedNatural`), tipos relacionados con colecciones (`Collection`, `Set`, `OrderedSet`, `Bag` y `Sequence`) y otros tipos de datos especiales. Por otro lado, OCL define también operaciones disponibles que se pueden realizar con cada uno de los tipos de datos: `and`, `or`, `not`, `+`, `-`, `<`, `>`, `union`, `size`, `includes`, etc.

La información proporcionada en el modelo UML con restricciones OCL es usada por el generador de propiedades para generar una especificación QuickCheck. Este componente utiliza la herramienta Dresden OCL [159, 160] para parsear el modelo UML y OCL y, de esta forma, poder generar la especificación QuickCheck. Por su parte, a partir de la especificación QuickCheck se generará un conjunto de casos de prueba abstractos (ATS), que podrán ser ejecutados gracias al uso del adaptador. La generación de casos de prueba a partir de la especificación QuickCheck depende de qué tipo de especificación se genere: propiedades independientes o una máquina de estados QuickCheck.

La decisión de si se usan propiedades independientes o máquinas de estados depende de la especificación del sistema a probar, es decir, del modelo. Así, si el modelo describe un componente sin estado, se generarán propiedades independientes, mientras que si el modelo describe un componente con estado se generará el código de una máquina de estados QuickCheck. En cualquier caso, la ejecución de los casos de prueba generados a partir de la especificación QuickCheck se realizará usando un adaptador, de la misma forma que se indica en la sección 6.4.1.5.

6.5.2. Componentes sin estado

En este trabajo se consideran componentes sin estado aquellos que no tienen un estado interno que se modifique con la ejecución de las operaciones que ofrece su API de integración. Así, las operaciones a probar en los componentes sin estado no tienen efectos colaterales, y son independientes entre sí. De esta forma, el resultado de invocar estas operaciones no depende de cuándo sean ejecutadas, sino que únicamente dependen de sus argumentos de entrada. Este tipo de componentes no suelen ser habituales en el mundo del software, pero existen algunos casos típicos que suelen funcionar de esta manera, como son, por ejemplo, ciertos tipos de librerías, como normalmente ocurre con las librerías matemáticas.

Como se ha comentado, el generador de propiedades utiliza un diagrama de clases y restricciones OCL que complementan la información mostrada en dicho diagrama de clases. El diagrama de clases se utiliza para conocer las operaciones que deben ser probadas, junto con los parámetros de entrada y el valor de retorno de cada una de ellas, conociendo los tipos de datos involucrados. Por otra parte, las restricciones OCL se usan para expresar precondiciones y postcondiciones para cada una de las operaciones a probar, así como invariantes de clase, es decir, condiciones que deben ser siempre ciertas cuando el estado del objeto se considere correcto.

Cuando el generador de propiedades actúa sobre un componente sin estado, se genera una propiedad por cada operación a probar. En concreto, para cada operación a probar op que recibe p parámetros $p^1, \ldots, p^p$, se genera una propiedad como la siguiente:

$$\forall\, \overline{p} \in G,\ pre(\overline{p}) \Rightarrow\ [r = op(\overline{p}), post_i(\overline{p}, r)\ \forall i \in [1, M]]$$

donde:

- $\overline{p}$ son los parámetros de entrada p^1,..., p^p de la operación op,
- p es el número de parámetros,
- G son los generadores de datos G^1,..., G^p para generar los argumentos de la operación, esto es, los valores de los parámetros p^1,..., p^p,
- pre son las precondiciones asociadas a la operación, que se corresponden con las precondiciones especificadas en OCL para esta operación,

- $post$ son las M postcondiciones asociadas a la operación ($post^1$, $post^2$, ..., $post^M$), las cuales se corresponden con las propias postcondiciones especificadas en OCL, las invariantes de clase y las restricciones de cuerpo asociadas a la operación.

Esta propiedad se escribe declarativamente en QuickCheck de la siguiente manera:

```
prop_op() ->
 ?FORALL(P, G(), ?IMPLIES(pre(P),
  begin
   R = f(P),
   post^1(P, R) andalso
      post^2(P, R) andalso
      ⋮
      post^M(P, R)
  end)).
```

Cuando QuickCheck ejecuta una propiedad como esta, la propia herramienta QuickCheck generará múltiples argumentos de entrada aleatorios (valores de los parámetros p^1,..., p^p), filtrando aquellos que no cumplen las precondiciones pre. Después de eso, se invoca la operación op con todos los argumentos de entrada válidos generados en el paso anterior, y comprobando si las postcondiciones $post$ son ciertas para cada ejecución. Este proceso se debe realizar para cada una de las operaciones a probar de todos los componentes a probar. Por tanto, en general, el proceso a seguir es el siguiente (ver figura 6.13):

1. Se selecciona un componente del modelo UML de entrada. Como los componentes no tienen estado, el orden de elección de componentes no es importante.

2. Se selecciona una operación del componente elegido. El orden de selección de las operaciones tampoco es importante, puesto que el componente no tiene estado interno que altere el resultado de las operaciones.

3. Se generan los argumentos de entrada para ejecutar la operación elegida.

4. Las precondiciones, si existen en la especificación OCL, se usan para generar estos argumentos de entrada:

   a) Si las precondiciones se cumplen, entonces la operación se ejecuta con los argumentos generados,

   b) En otro caso, se vuelve al paso 3.

5. Se comprueban las postcondiciones, si existen en OCL como postcondiciones, restricciones de cuerpo o invariantes.

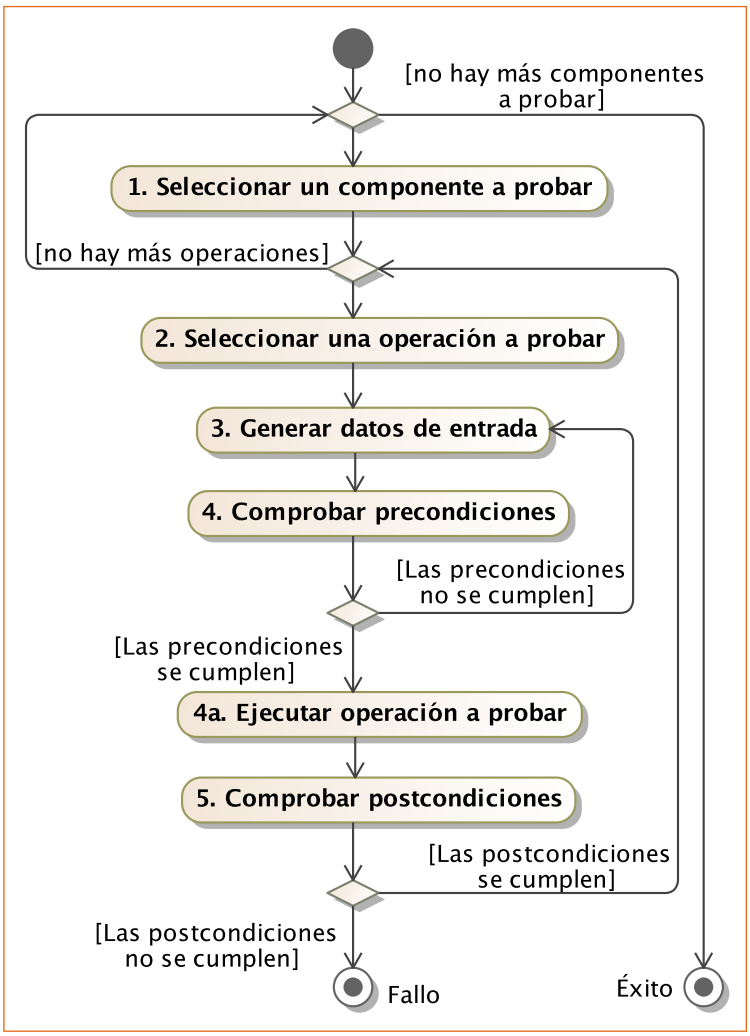


FIGURA 6.13: *Proceso a seguir para realizar las pruebas de componentes sin estado a partir de una especificación UML y OCL*

a) Si las postcondiciones se cumplen, se vuelve al paso 2.

b) En otro caso, el proceso de prueba se interrumpe, puesto que se ha producido un fallo. Si esto ocurre, QuickCheck ayuda a encontrar la causa de dicho fallo gracias al proceso de reducción que devuelve un caso de prueba más pequeño, a partir del caso de prueba original, que también provoca que el sistema no se comporte de la manera esperada [87, 135, 374].

6. El proceso finaliza cuando todas las operaciones de todos los componentes se hayan probado.

En cualquier caso, para realizar la transformación de OCL a QuickCheck es necesario definir cómo se transforman cada uno de los elementos de OCL, es decir, los

tipos de datos y operaciones, a construcciones válidas en Erlang que permitan definir las propiedades de QuickCheck. A continuación se describe cómo se realizan estas transformaciones.

6.5.2.1. Tipos de datos OCL

OCL es un lenguaje tipado, por lo que cada uno de los parámetros de las operaciones definidas tienen un tipo de dato asociado, así como cada operación en sí, que devuelve valores de un determinado tipo de dato. En concreto, el lenguaje OCL define los siguientes tipos de datos:

- Tipos de datos primitivos: `Real`, `Integer`, `String`, `Boolean` y `UnlimitedNatural`.
- Tipos de datos de colecciones (`Collection(T)`), que describen una lista de elementos de un tipo de dato específico (`T`). OCL define diferentes clases de colecciones dependiendo de si pueden contener elementos duplicados o no, y de si están o no ordenados: `Set(T)`, `OrderedSet(T)`, `Bag(T)` y `Sequence(T)`.
- Enumeraciones, que definen un conjunto de literales que representan los valores posibles de la enumeración.
- Tuplas, que se componen de varios valores, y cada uno de ellos puede pertenecer a un tipo de dato diferente.
- Otros tipos de datos especiales, como, por ejemplo, `OclAny`, `OclMessage`, `OclVoid` o `OclInvalid`.

Los generadores de datos G deben tener en cuenta los tipos de datos de los parámetros p^j de la operación a probar, op, para que puedan generar valores del tipo de dato correspondiente. Por tanto, se debe establecer una correspondencia entre los tipos de datos OCL y los tipos de datos Erlang que se usarán en las propiedades QuickCheck (ver tabla 6.2). Además, es necesario definir cómo generar valores del tipo de dato correspondiente usando un generador QuickCheck. Para ello, en QuickCheck, existen multitud de generadores de datos que se pueden usar directamente para este propósito. Así, por ejemplo, se usarán los siguientes generadores del módulo `eqc_gen`:

- `real` para generar valores con tipo de dato `Real`,
- `int` para generar valores con tipo de dato `Integer`,
- `nat` para generar valores con tipo de dato `UnlimitedNatural`,
- `bool` para generar valores con tipo de dato `Boolean`,
- `list` para generar valores con tipo de dato `Sequence`, el cual recibe como parámetro otro generador, `gen_T`, que genera valores del tipo `T`.

Tipo de dato OCL	Tipo de dato Erlang	Generador QuickCheck
`Real`	`float()`	`real()`
`Integer`	`integer()`	`int()`
`String`	`string()`	`list(char())`
`Boolean`	`boolean()`	`bool()`
`UnlimitedNatural`	-	`nat()`
`Set(T)`	Módulo `sets`	Específico
`OrderedSet(T)`	`list()`	Específico
`Bag(T)`	`list()`	Específico
`Sequence(T)`	`list(T)`	`list(gen_T())`
Enumeraciones	`tuple()`	Específico
Tuplas	`tuple()`	Específico
`OclAny`	-	Específico
`OclMessage`	-	-
`OclVoid`	`atom()`	Específico
`OclInvalid`	`atom()`	Específico

TABLA 6.2: *Tipos de datos OCL y generadores QuickCheck*

Por otro lado, es necesario definir generadores específicos para generar valores de los tipos de datos correspondientes a algunos de los tipos de datos OCL. Por ejemplo, para generar valores con tipo `Set(T)`, se usará el siguiente generador:

```
gen_set_T() ->
  ?LET(L, eqc_gen:list(gen_T()), sets:from_list(L)).
```

donde `gen_T` genera valores del tipo de dato `T`. Todos los generadores necesarios para generar valores del tipo de dato correspondiente OCL se han incluido en un módulo Erlang llamado `ocl_gen`, el cual se usará en los ejemplos posteriores.

Además, en algunas ocasiones, también es necesario generar objetos de una determinada clase definida por el usuario para que sean usados como argumentos de entrada en las operaciones a probar. En estos casos también es necesario definir generadores específicos. Para ello, la representación elegida para una instancia de una clase UML es una tupla cuyo segundo elemento es una lista de tuplas Erlang, donde, a su vez, cada tupla está compuesta por el nombre de un atributo de la clase y su valor correspondiente. Así, dada una clase Y con N atributos $a_1, a_2, \ldots, a_N$, con tipos de datos $d_1, d_2, \ldots, d_n$, la representación abstracta de una instancia de esta clase es:

$$\{Y, [\{a_1, v_1\}, \{a_2, v_2\}, \ldots, \{a_N, v_N\}]\}$$

donde:

- Si d_i es un tipo de dato simple, v_i representa el valor del atributo a_i.
- Si d_i es un tipo de dato complejo, v_i es una tupla con la siguiente forma, $\{d_i, [\{a_i^1, v_i^1\}, \{a_i^2, v_i^2\}, \ldots, \{a_i^M, v_i^M\}]\}$, donde a_i^j representa el atributo j de la clase d_i, y v_i^j su valor, el cual puede ser de nuevo un valor simple o bien una tupla.

Así, el generador de instancias de una clase compleja Y se implementa a través de una función como la siguiente:

```
gen_y() -> {Y, [{a1, g1()}, {a2, g2()}, ..., {aN, gN()}]}.
```

donde a_i es el nombre del atributo i de la clase Y, y g_i es un generador de valores del tipo de dato d_i. De esta forma, es posible generar instancias aleatorias de la clase Y con la representación abstracta elegida.

6.5.2.2. Operaciones OCL

El lenguaje OCL define operaciones predefinidas para cada uno de los tipos de datos, por ejemplo, +, -, *, `abs`, `size` (de una colección), etc. Estas operaciones, obviamente, también deben ser traducidas a código Erlang. Un gran número de operaciones definidas por OCL ya existen en el lenguaje Erlang, sin embargo, hay otras operaciones que no tienen una traducción directa.

Del mismo modo que el módulo `ocl_gen`, se han definido módulos Erlang para cada uno de los tipos de datos predefinidos en el lenguaje OCL (`ocl_int`, `ocl_seq`, `ocl_bag`, etc.), de tal forma que cada módulo incluye la implementación en Erlang de cada operación disponible en la especificación OCL. De esta forma, se mantienen las propiedades generadas lo más simple posible, y se mantienen los nombres de las operaciones en OCL para facilitar la comprensión de las mismas en caso de ser necesario acceder a ellas.

Por ejemplo, el módulo `ocl_seq`, que implementa todas las operaciones definidas en OCL para el tipo de dato `Sequence`, contiene, entre otras, la implementación de las siguientes operaciones:

```
including(Obj, Seq) ->
 Seq ++ [Obj].

excluding(Obj, Seq)->
 case lists:member(Obj, Seq) of
  true -> excluding(Obj, lists:delete(Obj, Seq));
  false -> Seq
 end.
```

donde la operación OCL `including(obj:T):Sequence(T)` devuelve un valor del tipo de dato `Sequence` que contiene todos los elementos de la lista original junto con el nuevo objeto `obj` añadido como último elemento; y la operación `excluding(obj:T):Sequence(T)` devuelve una secuencia que contiene todos los elementos excepto el objeto `obj` que se pasa como parámetro. Así, el siguiente código OCL:

```
l1->including(o)->excluding(p)
```

se traduce en Erlang de la siguiente forma:

```
ocl_seq:excluding(P, ocl_seq:including(O, L1)).
```

Como se puede observar, en el proceso de transformación se deben seguir las reglas básicas de la sintaxis de Erlang, como, por ejemplo, comenzar con una letra mayúscula los nombres de las variables. Aún así, a parte de esos detalles sintácticos, se consigue una traducción muy directa.

6.5.2.3. Iteradores OCL

Además de las operaciones descritas en la sección 6.5.2.2, el lenguaje OCL soporta iterar sobre todos los elementos de un objeto de tipo `Collection`. La operación de iteración más genérica que el lenguaje OCL define es `iterate`, que devuelve el valor de un acumulador que tiene un valor inicial que se actualiza usando una expresión evaluada para cada uno de los elementos de la colección. Sin embargo, el lenguaje OCL define otras operaciones de iteración sobre colecciones, como, por ejemplo, `select`, `reject`, o `collect`, entre otras, aunque todas ellas pueden ser sustituidas por el uso de la operación `iterate`.

De la misma forma que para las operaciones OCL descritas en la sección 6.5.2.2, los módulos Erlang relacionados con colecciones (`ocl_set`, `ocl_orderedset`, `ocl_bag` y `ocl_seq`), que contienen las operaciones para cada tipo de dato, también incluyen la implementación de los diferentes iteradores soportados para los tipos de datos `Collection`. Por ejemplo, el iterador `iterate` para el tipo de dato `Sequence` se implementa en el módulo `ocl_seq` de la siguiente forma:

```
iterate(F, Acc0, L)-> lists:foldl(F, Acc0, L).
```

Así, el siguiente ejemplo, que devuelve una nueva colección de tipo `Sequence` compuesta por el valor absoluto de todos los elementos de una colección de entrada también de tipo `Sequence`:

```
c->iterate(x: Integer;
   acc : Sequence(Integer) =
      Sequence(Integer){} |
         acc->including(x.abs()))
```

se traduce al siguiente código Erlang:

```
ocl_seq:iterate(
 fun(X, Acc) ->
  ocl_seq:including(Acc, ocl_int:abs(X))
 end, ocl_seq:new(), C)
```

donde la función `new` del módulo `ocl_seq` devuelve una nueva colección vacía de tipo `Sequence`:

```
new()-> [].
```

6.5.2.4. Expresiones OCL

Las expresiones definen la estructura del código OCL. Todas las expresiones OCL tienen un tipo, incluso si éste no se expresa explícitamente en muchas ocasiones. Ejemplos de expresiones son invocaciones a operaciones, accesos a propiedades de objetos, referencias a variables, expresiones literales, expresiones condicionales, etc. Las expresiones OCL pueden usarse, junto con las operaciones OCL, para definir diferentes tipos de restricciones, como son los valores iniciales, valores derivados, precondiciones, postcondiciones, invariantes, expresiones de cuerpo, o guardas, entre otras.

De la misma forma que las operaciones e iteradores OCL (ver secciones 6.5.2.2 y 6.5.2.3), las expresiones OCL también deben ser traducidas a código Erlang. Por ejemplo, la expresión condicional `if` de OCL se traduce a la estructura sintáctica `case` en Erlang.

Con respecto a las expresiones de navegación a través de las propiedades de los objetos, éstas se implementan usando una función Erlang que abstrae la representación interna de las instancias de clases como tuplas (ver sección 6.5.2.1). Esta función, llamada `get_prop`, se implementa de la siguiente forma:

```
get_prop(PropertyName, Variable) ->
 proplists:get_value(PropertyName, erlang:element(2, Variable)).
```

Por ejemplo, el siguiente código OCL:

```
manager.isUnemployed = false
```

se traduce a código Erlang de la siguiente forma:

```
ocl_bool:eq(get_prop(isUnemployed, Manager), false)
```

6.5.2.5. Ejemplo: librería de manipulación de listas

La figura 6.14 muestra un diagrama de clases UML que describe la API de acceso a una librería llamada `ListUtils`, la cual manipula listas de enteros y listas cuyos elementos son instancias de la clase `Item`. Para generar un conjunto de casos de prueba para este sistema, este modelo UML, junto con la especificación OCL, serán la entrada del generador de ATS.

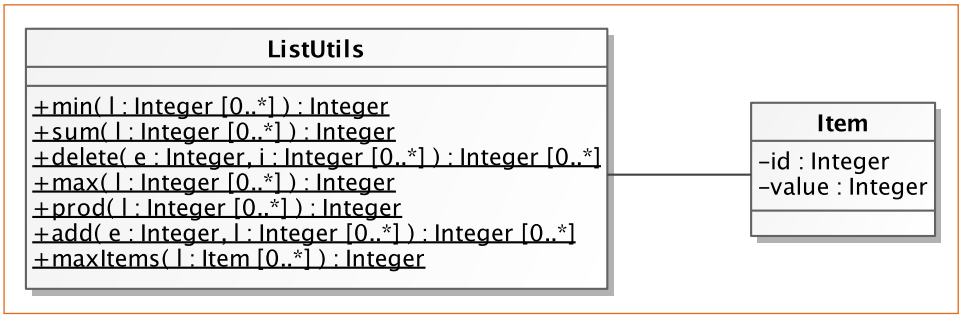


FIGURA 6.14: *Diagrama de clases UML de la librería de utilidad de listas*

Por ejemplo, la especificación OCL para la operación `max`, que devuelve el máximo número entero de una lista no vacía de enteros es la siguiente:

```
context ListUtils::max(l: Sequence(Integer)): Integer
pre not_empty: l -> notEmpty()
post check_max_integer: l -> forAll(x : Integer | result >= x)
post check_max_is_in_the_list: l->includes(result)
```

Para esta especificación, el generador de propiedades genera la siguiente propiedad QuickCheck:

```
prop_listutils_max()->
  ?FORALL(L, ocl_gen:gen_sequence_integer(),
    ?IMPLIES(
      begin
        ocl_seq:not_empty(L)
      end,
      begin
        Result = listUtils:max(L),
        ocl:tag([
          {"check_max_integer",
            ocl_seq:forAll(fun(X) -> (Result >= X) end, L)},
          {"check_max_is_in_the_list",
            ocl_seq:includes(Result, L)}])
      end)).
```

Como se puede observar, las postcondiciones especificadas en OCL se agrupan

en una única propiedad. Cada una de las postcondiciones se etiquetan asignándole un nombre, en este caso, el nombre usado en la postcondición en la especificación OCL, y usando la función `tag` del módulo `ocl`. Esta función devuelve `true` si todas las condiciones que se pasan como segundo elemento de las tuplas de la lista son ciertas, o, si alguna condición es `false`, el texto que se pasa como primer parámetro de la condición que falló:

```
tag([]) ->
 true;
tag([{_Name, true} | MoreTags]) ->
 tag(MoreTags);
tag([{Name, false} | _MoreTags]) ->
 Name.
```

Puesto que QuickCheck considera que la propiedad se cumple sólo si ésta devuelve el átomo `true`, cuando esto no sucede, se considera que la propiedad ha fallado, en cuyo caso se mostrará el resultado devuelto. Por tanto, en este caso, si la propiedad falla, se mostrará el nombre de la postcondición especificado en OCL que realmente no está cumpliendo la condición. Por tanto, el hecho de etiquetar las postcondiciones especificadas en OCL permite establecer una importante trazabilidad de la especificación en el resultado de las pruebas.

La propiedad anterior genera listas de enteros utilizando el generador de listas `gen_sequence_integer`. Las listas generadas que no satisfagan la precondición, es decir, `not_empty`, serán descartadas, utilizando únicamente listas válidas, esto es, no vacías, como argumento de entrada para ejecutar la operación `max` del módulo `listUtils`. Para cada ejecución, el resultado de la operación se almacena en una variable llamada `Result`, la cual se usará para comprobar si la postcondición es cierta.

Si se compara esta propiedad con la definida en la sección 5.4 (página 83), las cuales son equivalentes, puesto que ambas permiten probar la implementación de una función `max` que devuelve el máximo número entero de una lista no vacía de enteros, se observa como la propiedad mostrada en la sección 5.4, escrita manualmente, es más sencilla y simple de entender que la mostrada aquí. Por el contrario, esta propiedad, al ser generada automáticamente a partir de una especificación OCL, su estructura está basada en cómo se haya escrito dicha especificación OCL. De esta forma, aunque la propiedad pueda parecer más compleja, es posible establecer una relación directa entre la especificación OCL y los elementos Erlang usados en la propiedad gracias a la definición de los módulos por cada tipo de dato (como `ocl_seq`), y a la conservación de los nombres de las operaciones OCL en Erlang.

Por otro lado, un ejemplo más complejo es el de la operación `maxItems`, la cual devuelve el objeto con el máximo valor para la propiedad `value` de entre todos los elementos de la lista de entrada. En este caso, la lista no está formada por datos

simples como son los enteros (`Integer`), sino que está compuesta por elementos complejos (instancias de la clase `Item`). La especificación OCL de esta operación es la siguiente:

```
context ListUtils::maxItems(l: Sequence(Item)): Integer
pre not_empty: l->notEmpty()
post check_max_integer: l->forAll(x : Item | result >= x.value)
post check_max_is_in_the_list: l->select(value = result)->
 notEmpty()
```

En este caso, cuando el generador de propiedades analiza esta especificación OCL, genera la siguiente propiedad QuickCheck:

```
prop_max_check_maxItems() ->
 ?FORALL(L, gen_sequence_item()
  ?IMPLIES(
   begin
    ocl_seq:not_empty(L)
   end,
   begin
    Result = listUtils:maxItems(L),
    ocl:tag([
     {"check_max_integer",ocl_seq:forAll(
      fun(X) ->
       (Result >=
        ocl_datatypes:get_property(value, X))
       end, L)},
     {"check_max_is_in_the_list",
      ocl_seq:not_empty(ocl_seq:select(
       fun(ImplicitVariable0) ->
        (ocl_datatypes:get_property
          (value, ImplicitVariable0) ==
            Result)
       end, L)}])
   end)).
```

Cuando QuickCheck ejecuta esta propiedad, genera listas aleatorias de elementos de la clase `Item` usando el generador de datos `gen_sequence_item`:

```
gen_sequence_item() ->
 ocl_gen:gen_sequence(gen_item()).
```

donde `gen_item` genera una instancia de la clase `Item` usando el siguiente generador:

```
gen_item() ->
  {item, [{id, ocl_gen:gen_integer()},
    {value, ocl_gen:gen_integer()}]}.
```

Por ejemplo, a continuación se muestra algunas listas generadas durante la ejecución de la propiedad `prop_max_checkmax`:

```
[]
[{item,[{id,0},{value,0}]}]
[{item,[{id,-1},{value,0}]}]
[{item,[{id,1},{value,0}]}]
[{item,[{id,4},{value,2}]}, {item,[{id,-3},{value,2}]}]
[{item,[{id,0},{value,0}]}]
[{item,[{id,1},{value,-2}]}, {item,[{id,2},{value,-3}]}]
...
```

La primera lista generada por QuickCheck es la lista vacía, la cual, como no satisface la precondición es, por tanto, descartada. Las otras listas generadas sí satisfacen la precondición, por lo que se usarán para comprobar la postcondición. De esta forma, la operación `maxItems` se ejecuta sólo con listas válidas, comprobando que la postcondición sea cierta para cada una de ellas.

6.5.3. Componentes con estado

A diferencia de los componentes sin estado, los componentes con estado tienen un estado interno que puede ser modificado con la ejecución de las operaciones que ofrecen. Además, el comportamiento de las operaciones y, por tanto, el resultado esperado de cada una de ellas, puede depender, además de los argumentos recibidos, también de la información almacenada en este estado interno. Por tanto, en este caso, el estado del componente se debe tener en cuenta para las pruebas y, de esta forma, las precondiciones y postcondiciones dependen de dicho estado. Además, los casos de prueba deben estar formados por diferentes secuencias de operaciones, puesto que ahora el orden de ejecución de las mismas sí que puede variar el resultado esperado.

Los pasos a seguir para probar este tipo de componentes siguiendo esta aproximación son similares a los mostrados para los componentes sin estado, es decir, generar datos de entrada que se correspondan con cada uno de los parámetros de una operación, comprobar las precondiciones para dicha operación con dichos argumentos de entrada (y el estado actual), y, si se cumplen, ejecutarla con los datos de entrada generados para, finalmente, comprobar las postcondiciones para dicha operación (teniendo en cuenta los datos almacenados en el estado).

La diferencia con respecto a los componentes sin estado radica en que en este tipo de componentes es necesario almacenar cierta información en el propio módulo

de pruebas como un estado interno del mismo, puesto que las precondiciones y postcondiciones requieren estos datos adicionales para realizar las comprobaciones oportunas.

Por tanto, no es suficiente con definir propiedades independientes para cada una de las operaciones. En este caso, se generará una máquina de estados QuickCheck a partir de la especificación UML y OCL. A continuación se describirán los pasos que se siguen para generar automáticamente esta máquina de estados QuickCheck. Para ello se utilizarán dos ejemplos: un planificador de tareas [114, 316] y la implementación de un tipo de dato *pila* que almacena números enteros. El motivo para usar dos ejemplos diferentes es que, de esta forma, se pueden cubrir diferentes situaciones posibles que pueden ocurrir en este proceso.

La figura 6.15 muestra el diagrama de clases UML para el planificador de tareas que se usará como ejemplo. En este diagrama se muestran dos clases, `Scheduler`, que representa la API de integración del componente, y `Process`, que son las entidades que maneja el planificador. La especificación OCL asociada a cada una de las operaciones es la siguiente[2] :

```
context Scheduler
inv invariant: (ready->intersection(waiting))->isEmpty()
   and not ((ready->union(waiting))->includes(active))
   and (active = null implies ready->isEmpty())

context Scheduler::init()
post init: (ready->union(waiting)) = Set{} and active = null

context Scheduler::new(p:Process): Process
pre: p <> active
   and not (ready->union(waiting))->includes(p)
post new: waiting = (waiting@pre->including(p))
    and ready = ready@pre and active = active@pre
post new_result: result = p

context Scheduler::ready(p:Process): Process
pre: waiting->includes(p)
post ready: waiting = waiting@pre->excluding(p)
    and if active@pre = null then
          (ready = ready@pre and active = p)
      else (ready = ready@pre->including(p)
          and active = active@pre)
      endif
post ready_result: result = p
```

Por su parte, el tipo de dato *pila* se especifica a partir del diagrama de clases UML

[2] La especificación OCL para la operación `swap` se mostrará y analizará en la sección 6.5.4.

de la figura 6.16, y tiene la siguiente especificación OCL:

```
context Stack::pop(): Integer
pre notEmpty: isEmpty() = false
post topElementReturned: result = top@pre()
post elementRemoved: size() = size@pre() - 1

context Stack::top(): Integer
pre notEmpty: isEmpty() = false

context Stack::push(o: Integer)
post pushedObjectIsOnTop: top() = o
```

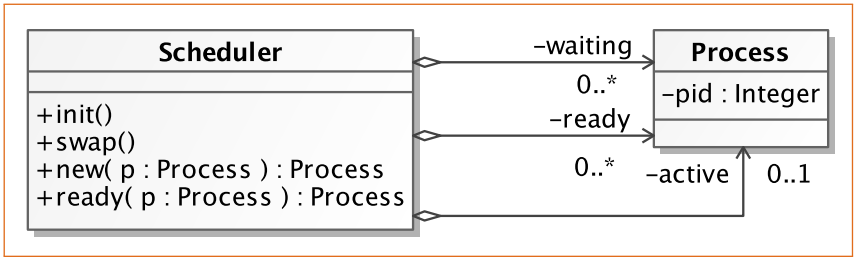


FIGURA 6.15: *Diagrama de clases UML del planificador de procesos*

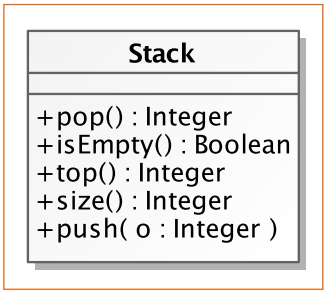


FIGURA 6.16: *Diagrama de clases UML del tipo de dato pila*

6.5.3.1. Definición del estado

Una de las peculiaridades del lenguaje OCL es la capacidad para expresar en las postcondiciones el valor que tenía una propiedad o, incluso, el resultado de invocar una operación antes de ejecutar la operación asociada. Para esto, el lenguaje OCL proporciona el operador `@pre`. De hecho, esta es la diferencia principal entre los componentes sin estado y los componentes con estado y, por tanto, será el criterio a utilizar para generar propiedades independientes o una máquina de estados QuickCheck.

Como se puede observar en el ejemplo del programador de tareas, el operador `@pre` se usa tanto en la postcondición de la operación `new` como en la postcondición

de la operación `ready`, en ambos casos para recuperar el valor de una propiedad antes de ejecutar la operación `new` o `ready`. Puesto que se necesitan conocer los valores a los que hace referencia la operación `@pre` antes de ejecutar una operación en las pruebas, éstos se almacenarán en el estado del módulo de pruebas. Así, en este ejemplo, el estado almacenará los valores de las propiedades `active`, `ready` y `waiting`:

```
-record(ts, {active, ready, waiting}).
```

En la especificación de las operaciones de la clase `Stack`, el uso del operador `@pre` es ligeramente diferente, puesto que se aplica al resultado de invocar operaciones, en vez de recuperar valores de propiedades. Estos valores también deben almacenarse en el estado. Así, por ejemplo, para comprobar el resultado de la operación `pop` se requiere conocer el resultado de las operaciones `top` y `size` antes de ejecutar la propia operación `pop`. Es por ello que en el estado se deben almacenar estos dos valores.

De esta forma, la aproximación seguida es almacenar en el estado todas aquellas propiedades o resultados de operaciones para las cuales se requiere el valor anterior a una ejecución, esto es, que aparecen en alguna postcondición acompañadas del operador `@pre`. Además, también se almacenarán en estado los valores correspondientes a invocaciones a operaciones en las precondiciones y en las postcondiciones. Así, se evita realizar llamadas a operaciones desde las precondiciones o postcondiciones de QuickCheck, haciendo que éstas únicamente dependan del valor almacenado en el estado para cada operación.

Por ejemplo, en el caso de la precondición asociada a la operación `pop`, se realiza una llamada a la operación `isEmpty`, y, de forma similar, en la postcondición de la operación `push` se realiza una llamada a la operación `pop`. Estos valores también serán añadidos al estado, junto con el valor de `size` y el propio `top` para los cuales se usa el operador `pre`. Así, la definición del estado para este ejemplo es la siguiente:

```
-record(ts, {isEmpty, size, top}).
```

6.5.3.2. Inicialización del estado

Como se ha comentado en la sección 6.5.3.1, el estado está formado por los siguientes elementos:

1. Propiedades para las que se requiere el valor previo a ejecutar una operación usando el operador `@pre`.

2. Resultados de invocaciones a operaciones utilizando el operador `@pre`.

3. Resultados de invocaciones a operaciones que se requieran en las precondiciones.

4. Resultados de invocaciones a operaciones que se requieran en las postcondiciones.

Para las propiedades (caso 1), la aproximación seguida para inicializar el estado es consultar el valor de las mismas en el sistema a probar. De esta forma, en el adaptador, además de implementar cómo se deben invocar las operaciones en el sistema a probar, es necesario implementar cómo obtener de dicho sistema los valores de estas propiedades almacenadas en el estado (en caso de que estas operaciones no existieran en la API del componente). Por ejemplo, esta es la inicialización del estado para el `Scheduler`:

```
initial_state()-> #ts {
 active = scheduler:get_active(),
 ready = scheduler:get_ready(),
 waiting = scheduler:get_waiting()}.
```

Esto implica que en el módulo adaptador, llamado `scheduler`, se deben incluir implementaciones para las funciones `get_active`, `get_ready` y `get_waiting`, las cuales deben devolver en la notación abstracta entendible por el módulo de pruebas cuál es el proceso marcado como `active`, los procesos `ready` y los procesos `waiting` respectivamente.

Para los resultados de invocaciones a operaciones (casos 2, 3 y 4), en cambio, no es posible seguir esta aproximación, puesto que las operaciones podrían tener asociadas a su vez precondiciones que no permitan su invocación en el momento de necesitar obtener el estado actual. En este último caso, se opta por inicializar estos valores con un valor por defecto (`undefined`). Por ejemplo, la inicialización del estado para la clase `Stack` es la siguiente:

```
initial_state() -> #ts {
 isEmpty = undefined,
 size = undefined,
 top = undefined
}.
```

Por el contrario, como se mostrará más adelante, estos valores serán recuperados únicamente cuando sea necesario, en concreto, en las funciones envoltorio que se definen en el propio módulo de pruebas para invocar las operaciones del sistema a probar a través del adaptador.

6.5.3.3. Definición de las operaciones a probar

El generador de máquinas de estados debe identificar qué operaciones formarán parte de la misma. Esta información se obtiene a partir del diagrama de clases UML, en el cual se especifican, para la clase que representa la interfaz de acceso al componente, las operaciones que forman parte de dicha clase, junto con su nombre, parámetros de entrada y tipo de dato del valor de retorno. Con esta información es posible generar la función `command` de la máquina de estados, en la que se especifican las operaciones a probar y cómo se generan los argumentos para cada operación. Para producir estos argumentos se usan generadores, que a su vez son generados con las mismas técnicas que las explicadas en la sección 6.5.2.1.

Por ejemplo, para el programador de tareas, el generador de máquinas de estados produce automáticamente la siguiente función `command`, en la que se incluyen las tres operaciones definidas para la clase `Scheduler`: `init`, `new` y `ready`:

```
command(PreState) ->
  eqc_gen:oneof([
    {call, ?MODULE, init, []},
    {call, ?MODULE, new, [gen_process()]},
    {call, ?MODULE, ready, [gen_process()]}
  ]).
```

donde `gen_process` es, a su vez, un generador de instancias de la clase `Process`, las cuales se representan de forma abstracta como una tupla:

```
gen_process() -> {process, [{id, eqc_gen:int()}]}.
```

Por su parte, para la clase `Stack`, la función `command` generada tiene el siguiente aspecto:

```
command(PreState) ->
  eqc_gen:oneof([
    {call, ?MODULE, pop, []},
    {call, ?MODULE, push, [ocl_gen:gen_integer()]},
    {call, ?MODULE, top, []},
    {call, ?MODULE, size, []},
    {call, ?MODULE, isEmpty, []}
  ]).
```

Como se ha comentado en la sección 6.3.1.3, las operaciones que se llaman en la función `command` suelen ser funciones envoltorio definidas en el propio módulo de pruebas que invocan las operaciones del sistema a probar a través del adaptador, como ocurre en este caso. La implementación de estas funciones envoltorio producida por el generador de máquinas de estados sigue siempre la misma estructura:

- Inicialmente se recuperan aquellos valores del estado correspondientes a invocaciones a operaciones para las cuales se requiere el valor previo, es decir, el caso 2 descrito en la sección 6.5.3.2, quedando almacenado en la variable `DynPreState`. Por ejemplo, para la operación `pop` de la clase `Stack`, `DynPreState` debe contener el resultado de invocar a las operaciones `top` y `size`, puesto que éstos se necesitan antes de invocar la propia operación `pop`:

```
DynPreState = #ts {
   size = stack:size(),
   top = stack:top()
},
```

- En segundo lugar se ejecuta la operación a probar, la cual, en este caso, realizará una llamada a un adaptador externo que a su vez invocará al sistema a probar. Su valor se almacena en la variable `Result`. Por ejemplo:

```
Result = stack:pop(),
```

- Posteriormente, se recuperan aquellos valores del estado correspondientes con invocaciones a funciones realizadas en cualquiera de las precondiciones (caso 3 de la sección 6.5.3.2), así como las invocaciones a las operaciones que se requieran en las postcondiciones de la propia operación (caso 4). Este estado se almacena en la variable `DynAfterState`. Por ejemplo, la operación `pop` necesita obtener el valor de la operación `isEmpty`, puesto que está en las precondiciones de alguna operación de la especificación OCL, y el valor de la operación `size`, puesto que se requiere en la propia postcondición de la operación `pop`:

```
DynAfterState = #ts {
   isEmpty = stack:isEmpty(),
   size = stack:size()
},
```

  Como se explicará posteriormente en la sección 6.5.4, para aquellos componentes parcialmente especificados también es necesario recuperar aquellos valores del estado correspondientes a propiedades (caso 1 descrito en la sección 6.5.3.2).

- Finalmente se devuelve una tupla con los valores:

```
{Result, DynPreState, DynAfterState}
```

Así, la implementación de la función envoltorio que invocará la operación `pop` de la clase `Stack` es la siguiente:

```
pop() ->
 DynPreState = #ts {
  size = stack:size(),
  top = stack:top()
 },
 Result = stack:pop(),
 DynAfterState = #ts {
  isEmpty = stack:isEmpty(),
  size = stack:size()
 },
 {Result, DynPreState, DynAfterState}.
```

Para las operaciones del ejemplo del `Scheduler`, los valores de `DynPreState` y `DynAfterState` no son necesarios en las precondiciones y postcondiciones, por lo que la implementación de las operaciones serán todas del siguiente estilo:

```
new(P)->
 DynPreState = #ts { },
 Result = scheduler:new(P),
 DynAfterState = #ts { },
 {Result, DynPreState, DynAfterState}.
```

6.5.3.4. Evolución de la información almacenada en el estado

La evolución de los datos almacenados en el estado se define a través de la función `next_state`. La generación de esta función es la parte más complicada de este proceso, puesto que se necesita predecir cómo cambian los datos del estado con la ejecución de cada una de las operaciones, únicamente utilizando la información de las postcondiciones e invariantes de la especificación OCL.

Así, por ejemplo, la postcondición de la operación `new` de la clase `Scheduler` indica que después de ejecutar esta operación, a los procesos almacenados en la variable `waiting` se les añadirá el nuevo proceso que se pasa como argumento a la operación `new`, y los procesos `ready` y `active` no cambiarán. Esto implica que la función `next_state` generada para la operación `new` sea la siguiente:

```
next_state(PreState, Result, {call, ?MODULE, new, [P]}) ->
 PreState#ts {
  waiting = ocl_set:including(P, PreState#ts.waiting),
  ready = PreState#ts.ready,
  active = PreState#ts.active
 };
```

Por tanto, este proceso de generación de funciones `next_state` consiste en considerar los operadores de igualdad (=) sobre propiedades del estado como asigna-

ciones. Además, se debe tener en cuenta que únicamente es posible considerar las expresiones deterministas para predecir el valor que va a tener el estado después de la ejecución de una operación. Por ejemplo, si `A` y `B` son dos propiedades almacenadas en el estado, y una expresión OCL tiene la siguiente forma:

```
A = 1 and B = 2
```

es posible construir la función `next_state`, puesto que después de ejecutar la operación `A` siempre deberá tener el valor `1` y `B` el valor `2`. En cambio, una expresión del siguiente tipo:

```
A = 1 or B = 2
```

no permite construir la función `next_state` correctamente. De la misma forma, expresiones como esta:

```
if (X = 1) then
   A = 1
else
   B = 2
end
```

tampoco permiten construir la función `next_state`. Estos últimos casos caen dentro de la categoría de componentes parcialmente especificados, los cuales se describen en la sección 6.5.4, puesto que no especifican totalmente cómo evolucionan las propiedades del estado a medida que se ejecutan las diferentes operaciones a probar. Por el contrario, si la especificación OCL describe cómo cambian las propiedades del estado en cada una de las situaciones posibles, como, por ejemplo:

```
if (X = 1) then
   A = 1 and B = B@pre
else
   B = 2 and A = A@pre
end
```

sí podría generarse correctamente la función `next_state`, y utilizarse esta aproximación.

En ocasiones, existen casos en los que se deben realizar transformaciones lógicas o algebraicas para averiguar cuál sería el próximo valor de una propiedad del estado después de ejecutar una operación. Por ejemplo, la postcondición de la operación `init` indica que la unión de los procesos `ready` y `waiting` es el conjunto vacío:

```
(ready->union(waiting)) = Set{}
```

pero esto quiere decir que tanto `ready` como `waiting` serán el conjunto vacío, es

decir:

```
ready = Set{} and waiting = Set{}
```

Por tanto, este tipo de transformaciones también deben tenerse en cuenta para generar la función `next_state`.

Cuando los valores almacenados en el estado no son propiedades, sino resultados de invocar a otras operaciones, esta aproximación no puede llevarse a cabo, puesto que no es posible predecir el valor del estado durante la primera pasada de ejecución de la máquina de estados QuickCheck, es decir, sin ejecutar las propias operaciones a probar. En este caso, se usarán los valores simbólicos devueltos por las funciones envoltorio como `DynAfterState`, los cuales representan los valores del estado una vez ejecutada la operación.

Puesto que la función `next_state` es ejecutada por QuickCheck tanto en la primera pasada, con valores simbólicos, como en la segunda pasada, con valores reales, el valor de `DynAfterState` y la información almacenada en dicha variable deben ser recuperados utilizando la notación simbólica. Así, por ejemplo, esta es la función `next_state` generada para la operación `pop` de la clase `Stack`:

```
next_state(PreState, Result, {call, ?MODULE, pop, []}) ->
 DynAfterState = {call, erlang, element, [3, Result]},
 PreState#ts {
  isEmpty = {call, erlang, element, [2, DynAfterState]},
  size = {call, erlang, element, [3, DynAfterState]},
  top = {call, erlang, element, [4, DynAfterState]}
 };
```

6.5.3.5. Precondiciones asociadas a cada operación

Las precondiciones QuickCheck para cada operación se obtienen a partir de las precondiciones asociadas a dicha operación en la especificación OCL. De esta forma, la precondición que se crea en la máquina de estados QuickCheck para cada operación es la conjunción de todas las precondiciones definidas para dicha operación en OCL.

El proceso de transformación de la especificación OCL a QuickCheck es exactamente el mismo que se explica en la sección 6.5.2, es decir, usando los módulos Erlang que contienen la implementación de las operaciones disponibles en OCL. Sin embargo, en este caso, es necesario tener en cuenta que la información requerida sobre propiedades o resultados de invocar operaciones son datos que ya han sido almacenados en el estado en la función `next_state`, por lo que se deben obtener de ahí. Por ejemplo, la operación `new` de la clase `Scheduler` tiene asociada la siguiente precondición en la máquina de estados QuickCheck:

```
precondition(PreState, {call, scheduler, new, [P]})->
  (ocl:neq(P, PreState#ts.active)
   andalso not(ocl_set:includes(P,
    ocl_set:union(PreState#ts.waiting, PreState#ts.ready))));
```

Como se observa, los valores de las propiedades `active`, `waiting` y `ready` se obtienen del estado `PreState`, puesto que es dónde están almacenados.

En el ejemplo referente a la clase `Stack`, la aproximación es la misma, es decir, cuando se requiere el valor de la operación `isEmpty`, ésta se recupera del estado, puesto que estará ahí almacenada. De todas formas, en este ejemplo, al no poder predecir cómo evoluciona el estado sin necesidad de ejecutar las operaciones a probar, no es posible usar las precondiciones estándar de QuickCheck, puesto que éstas se evalúan en una primera pasada, en la cual todavía no se ejecutan las operaciones de la máquina de estados. Por el contrario, se usarán precondiciones dinámicas, las cuales son evaluadas únicamente en la segunda pasada, cuando las operaciones a probar son realmente ejecutadas. Las precondiciones dinámicas se definen de una manera similar que las precondiciones estándar explicadas en la sección 6.3.1.5, salvo que el nombre de la función Erlang a definir es `dynamic_precondition` en vez de `precondition`:

```
dynamic_precondition(PreState, {call, ?MODULE, pop, []})->
  (PreState#ts.isEmpty == false);
```

Como se comentará más adelante, en la sección 6.5.4, en los componentes parcialmente especificados también será necesario utilizar este tipo de precondiciones dinámicas.

6.5.3.6. Postcondiciones asociadas a cada operación

Al contrario que ocurre con las precondiciones, las postcondiciones (e invariantes) especificadas en OCL, aunque también se agrupan a la hora de generar el código de la máquina de estados QuickCheck, se etiquetan usando la misma función `tag` implementada en el módulo `ocl` empleada en los componentes sin estado. De esta forma, si una postcondición de una operación falla, es posible conocer exactamente cuál de las postcondiciones especificadas en OCL para dicha operación realmente no se cumple.

Por lo demás, el acceso a propiedades y resultados de invocaciones a operaciones debe realizarse de la misma forma que en las precondiciones, es decir, recuperando los valores del estado. En este caso, se debe tener en cuenta si se requiere el valor antes de ejecutar la operación, o bien después de ejecutarla. Para facilitar y producir un código más legible y similar a la especificación OCL se define una función

`postcondition` genérica, que se encargará de obtener el estado siguiente utilizando la función `next_state` y, posteriormente, llamar a la postcondición específica de cada de las operaciones:

```
postcondition(PreState, C, R)->
 AfterState = eqc_symbolic:eval(next_state(PreState, R, C)),
 postcondition(PreState, AfterState, C, R).
```

Así, en la definición de la postcondición específica de cada operación estará disponible tanto el estado anterior como el estado siguiente, así como los estados dinámicos producidos en la función envoltorio correspondiente. Por ejemplo, esta es la postcondición generada para la operación `new`:

```
postcondition(PreState, AfterState, {call, ?MODULE, new, [P]},
   {Result, DynPreState, DynAfterState})->
 ocl:tag([
   {"new", (((ocl_set:eq(AfterState#ts.waiting,
         ocl_set:including(P, PreState#ts.waiting)) andalso
      ocl_set:eq(AfterState#ts.ready, PreState#ts.ready)) andalso
      ocl:eq(AfterState#ts.active, PreState#ts.active))},
   {"new_result", (ocl:eq(Result, P)))}
   {"invariant", invariant(PreState, AfterState,
      {call, ?MODULE, new, [P]},
      {Result, DynPreState, DynAfterState})}]);
```

donde `invariant` es una función automáticamente generada siguiendo la misma aproximación que para las postcondiciones.

Como se observa, las propiedades accedidas en la especificación OCL mediante el operador `@pre` son recuperadas del estado anterior (`PreState`), mientras que el resto de propiedades se recuperan del estado posterior a la ejecución de la operación `new` (`AfterState`).

Cuando no es posible predecir cómo evoluciona el estado sin necesidad de ejecutar las operaciones a probar, como ocurre con el ejemplo de la clase `Stack`, es necesario usar el valor `DynPreState`, devuelto por las funciones envoltorio, para acceder al estado. Por ejemplo, esta es la postcondición para la operación `pop` de la clase `Stack`:

```
postcondition(PreState, AfterState, {call, ?MODULE, pop, []},
   {Result, DynPreState, DynAfterState})->
 ocl:tag([{"topElementReturned", (Result == DynPreState#ts.top)},
       {"elementRemoved",
        (AfterState#ts.size == (DynPreState#ts.size - 1))}]);
```

6.5.3.7. Propiedad a comprobar

La propiedad que se genera para las máquinas de estados es común a cualquier especificación. Cabe destacar la necesidad de definir una operación `init_sut` en el adaptador usado para conectar el módulo de pruebas con el sistema a probar. Esta operación debe inicializar el sistema a probar, estableciéndole un estado en el cual se puedan comenzar a realizar las pruebas. Por ejemplo, esta es la propiedad para la clase `Scheduler`:

```
prop_state_machine() ->
 ?SETUP(fun() -> scheduler:init_sut(), fun() -> ok end end,
 ?FORALL(Cmds, eqc_statem:commands(?MODULE),
  begin
   {H, S, Res} = eqc_statem:run_commands(?MODULE, Cmds),
   scheduler:init_sut(),
   eqc_statem:pretty_commands(?MODULE, Cmds, {H, S, Res},
     Res == ok)
  end)).
```

La ejecución de esta propiedad provocará una generación de secuencias válidas de operaciones definidas en la función `command` teniendo en cuenta las precondiciones asociadas y, posteriormente, en una segunda pasada, ejecutará las secuencias generadas, comprobando las precondiciones y postcondiciones.

La ejecución de las secuencias de operaciones en el ejemplo del planificador de tareas `Scheduler` es diferente al ejemplo de la clase `Stack`. En la clase `Scheduler` únicamente se generarán secuencias válidas, es decir, secuencias en las que cada operación a ser ejecutada cumple sus precondiciones como, por ejemplo, la mostrada en la figura 6.17. De esta forma, si al ejecutar esta secuencia alguna operación no cumple sus precondiciones asociadas (o sus postcondiciones) se informará de un error y se parará la prueba, puesto que quiere decir que algo erróneo ha ocurrido.

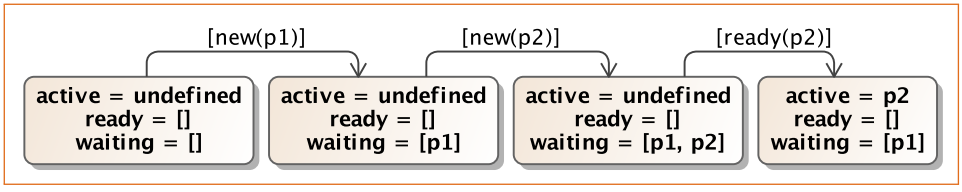


FIGURA 6.17: *Secuencia de comandos de ejemplo para probar la clase `Scheduler`*

En cambio, para la clase `Stack`, el comportamiento es diferente, puesto que no se usarán las precondiciones en la fase de generación, ya que éstas dependen de valores almacenados en el estado que no se pueden calcular sin ejecutar las operaciones. Es por ello que se generarán secuencias de operaciones que, al ser ejecutadas, podría darse el caso de que alguna de las operaciones que forme parte de esta secuencias sea descartada y no sea ejecutada por no cumplir las precondiciones dinámicas de-

finidas. Este es el caso de la operación `pop`, diferenciada en la figura 6.18. En este ejemplo, aunque la operación se incluye en la secuencia de operaciones a ejecutar, puesto que no se comprueban las precondiciones en el proceso de generación, en la segunda pasada, cuando estas secuencias se ejecutan y se utiliza el estado dinámico devuelto por las funciones envoltorio definidas en el módulo de pruebas, esta operación se descarta, puesto que la precondición dinámica (`isEmpty() = false`) no se cumple.

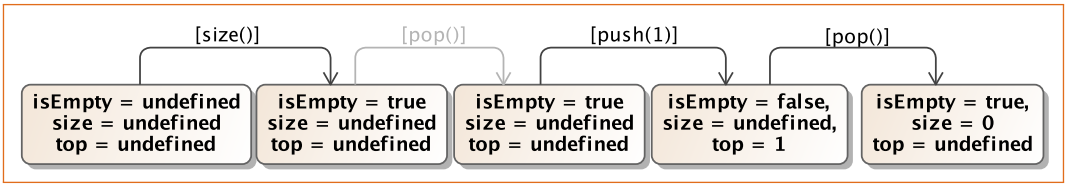


FIGURA 6.18: *Secuencia de comandos de ejemplo para probar la clase* `Stack`

6.5.4. Componentes parcialmente especificados

Existen situaciones en las que hay que tratar con especificaciones parciales o incompletas. Este hecho debe tenerse en cuenta en la generación de la máquina de estados QuickCheck, puesto que puede alterar dicho proceso de generación. Por ejemplo, añadir la siguiente especificación OCL para la operación `swap` convertirá al planificador de tareas en un componente parcialmente especificado:

```
context Scheduler::swap()
pre: active <> null
post: if ready@pre->isEmpty() then
      (active = null and ready = Set{})
    else (ready@pre->includes(active) and
        ready = ready@pre->excluding(active))
    endif and
    waiting = waiting@pre->including(active@pre)
```

La operación `swap` selecciona un proceso de la lista de procesos `ready` (si esta lista no está vacía) y lo establece como proceso activo, es decir, almacenándolo en la propiedad `active` (`ready@pre->includes(active)`), mientras que el anterior proceso almacenado en la propiedad `active` se añade a la lista de procesos `waiting` (`waiting = waiting@pre->including(active@pre)`). Sin embargo, esta especificación no indica qué proceso de la lista de procesos `ready` será seleccionado como el siguiente proceso activo. Así, el criterio de planificación de procesos no está completamente especificado. Por tanto, podrían existir diferentes posibles implementaciones (seleccionar un proceso al azar, seleccionar el primer proceso de la lista, seleccionar el proceso más antiguo, etc.) y todas ellas serían válidas de acuerdo a esta especificación.

En este caso, esta falta de especificación tiene como consecuencia que sea imposible generar la función `next_state` para la operación `swap`, puesto que es necesario ejecutar la propia operación `swap` para conocer el siguiente estado de pruebas, en concreto, el proceso marcado como `active`. Puesto que los valores almacenados en el estado se deben utilizar en las precondiciones, y éste no se puede calcular en la primera pasada de generación de secuencias de prueba sin ejecutar las operaciones a probar, la aproximación seguida es no utilizar las precondiciones con valores simbólicos en la etapa de generación de secuencias de operaciones. De esta forma, cuando el generador de máquinas de estados detecta esta situación, generará precondiciones que siempre se cumplen para todas las operaciones:

```
precondition(_S, _C)-> true.
```

Por el contrario, se filtrarán operaciones en la segunda etapa de la máquina de estados QuickCheck, es decir, en la etapa de ejecución. Para ello, se definirán precondiciones dinámicas, es decir, condiciones que serán evaluadas justo antes de decidir si una operación debe o no ser ejecutada. De esta forma, se puede considerar que las dos pasadas de la máquina de estados QuickCheck se mezclan en una única pasada, donde la secuencia de operaciones se genera *online*, es decir, la secuencia de operaciones se elige a medida que se van generando, en contraposición con la generación estándar *offline*, donde primeramente se genera una secuencia de operaciones válida, que será íntegramente ejecutada. Por ejemplo, para la operación `swap`, esta es la precondición dinámica generada:

```
dynamic_precondition(PreState, {call, ?MODULE, swap, []})->
 ocl:neq(PreState#ts.active, undefined);
```

En este tipo de componentes, las funciones `next_state` usarán el valor de la variable `DynAfterState`, devuelto por las funciones envoltorio, para establecer el estado siguiente que será usado por las postcondiciones, es decir, se generarán funciones `next_state` del siguiente estilo:

```
next_state(PreState, Result, {call, ?MODULE, swap, []}) ->
 DynAfterState = {call, erlang, element, [3, Result]},
 PreState#ts {
  active = {call, erlang, element, [2, DynAfterState]},
  ready = {call, erlang, element, [3, DynAfterState]},
  waiting = {call, erlang, element, [4, DynAfterState]}
 };
```

Por este motivo, las funciones envoltorio, descritas en la sección 6.5.3.3, deben devolver como `DynAfterState` también las propiedades almacenadas en el estado para este tipo de componentes parcialmente especificados. Por ejemplo:

```
swap()->
 DynPreState = #ts { },
 Result = scheduler:swap(),
 DynAfterState = #ts {
  active = scheduler:get_active(),
  ready = scheduler:get_ready(),
  waiting = scheduler:get_waiting()
 },
 {Result, DynPreState, DynAfterState}.
```

6.5.5. Caso de estudio: componente de gestión de contenidos multimedia

Basándose en el mismo ejemplo utilizado en la sección 6.4.2, que ilustra cómo usar la metodología de pruebas, es decir, la API de integración que proporciona el componente VoDKA Asset Manager, esta sección describe cómo ha sido el proceso de utilizar un modelo UML y OCL como especificación del sistema para realizar las pruebas.

Partiendo del diagrama de actividades de la figura 6.5 de la página 130, únicamente las fases de especificación de requisitos y la especificación de pruebas varía con respecto a la metodología de pruebas original descrita en la sección 6.4. Solamente se debe tener en cuenta que los adaptadores que conectan el ATS con el sistema a probar manejen la información esperada por nueva la especificación de pruebas QuickCheck, pero esto puede ser realizado de la misma forma que se describe en la sección 6.4.2.5.

Además, a diferencia de la aproximación original, ahora la especificación de los requisitos se realiza a través de un modelo UML con restricciones OCL que expresa cuáles son las operaciones de la API de integración y cuál es el comportamiento de cada una de ellas. Por otra parte, la especificación de pruebas ya no se escribe manualmente, sino que se genera automáticamente a partir del modelo UML con restricciones OCL.

Por tanto, la tabla 6.1 de la página 140, que muestra las operaciones de la API de integración del componente VoDKA Asset Manager, debe ser formalizada ahora como un modelo de clases UML. Este modelo de clases, mostrado en la figura 6.19, contiene la clase `AssetFacade` que representa la interfaz de acceso al sistema que define las operaciones a probar de la API de integración, es decir: `create`, `findById`, `findAll`, `update`, `delete`, `init` y `reset`. Además, también se puede observar que se han creado clases adicionales para representar los tipos de datos complejos que maneja el sistema, como es la clase `Asset`, así como cada uno de los errores posibles que pueden devolver cada una de las operaciones (`NotStarted`, `AuthenticationError`, `DuplicateAsset`, `NotFound`, etc.)[3] .

[3] Como se observa, se han cambiado ligeramente los nombres de las operaciones y los errores, por ejemplo, `findById` en vez de `find_by_id`, o `NotFound` en vez de `not_found`, para que los nombres utilizados sigan una nomenclatura más habitual en el lenguaje UML.

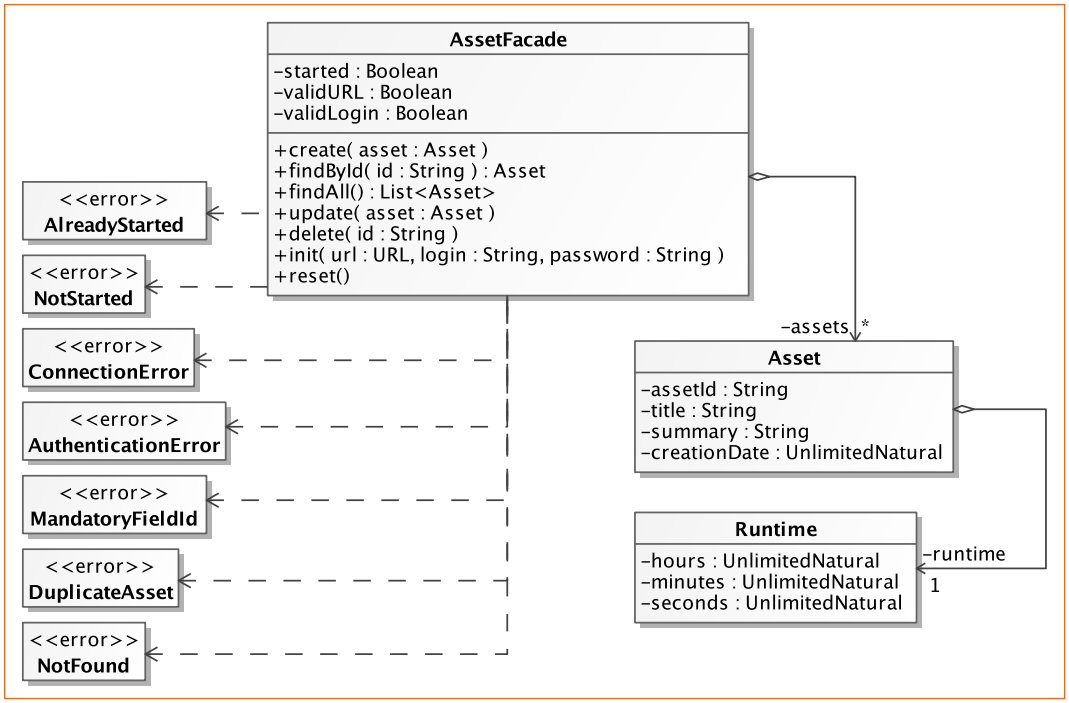


FIGURA 6.19: *Diagrama de clases UML de VoDKA Asset Manager*

Así, la información que representa el diagrama de clases de la figura 6.19 es exactamente la misma que la que se representa en la tabla 6.1. Sin embargo, es necesario especificar el comportamiento dinámico de cada operación. Esta información, que en la aproximación original se realizaba de manera informal, ahora se proporciona a través de restricciones OCL. Por ejemplo, la especificación OCL de la operación `create` es la siguiente:

```
context AssetFacade::create(asset:Asset)
post:
 if(self.started = false) then
  self.started = self.started@pre and
  self.validURL = self.validURL@pre and
  self.validLogin = self.validLogin@pre and
  self.assets = self.assets@pre and
  result.oclIsTypeOf(NotStarted)
 else
  if(self.validURL = false) then
   self.started = self.started@pre and
   self.validURL = self.validURL@pre and
   self.validLogin = self.validLogin@pre and
   self.assets = self.assets@pre and
   result.oclIsTypeOf(ConnectionError)
```

```
  else
   if(self.validLogin = false) then
    self.started = self.started@pre and
    self.validURL = self.validURL@pre and
    self.validLogin = self.validLogin@pre and
    self.assets = self.assets@pre and
    result.oclIsTypeOf(AuthenticationError)
   else
    if(asset.assetId = '') then
     self.started = self.started@pre and
     self.validURL = self.validURL@pre and
     self.validLogin = self.validLogin@pre and
     self.assets = self.assets@pre and
     result.oclIsTypeOf(MandatoryFieldId)
    else
     if (self.assets@pre -> select(a |
        a.assetId = asset.assetId) -> size() > 0) then
      self.started = self.started@pre and
      self.validURL = self.validURL@pre and
      self.validLogin = self.validLogin@pre and
      self.assets = self.assets@pre and
      result.oclIsTypeOf(DuplicateAsset)
     else
      self.started = self.started@pre and
      self.validURL = self.validURL@pre and
      self.validLogin = self.validLogin@pre and
      self.assets = self.assets@pre->including(asset)
     endif
    endif
   endif
  endif
endif
```

Esta especificación es la formalización en OCL de la descripción informal del comportamiento de la operación `create` que se muestra en la página 140, y se debe realizar para cada una de las operaciones a probar de la API de integración. Cabe mencionar que, para cada propiedad que forma parte del estado, es decir, los atributos `started`, `validURL`, `validLogin` y `assets`, se especifica cuál será su valor en referencia al valor que tenían antes de ejecutar la operación. De esta forma, se evita caer en el ámbito de componentes parcialmente especificados.

Esta especificación OCL, junto con el diagrama de clases UML, se usa para generar automáticamente la especificación de pruebas. En este caso, como se usa el operador `@pre` en la especificación OCL, la especificación de pruebas es una máquina de estados QuickCheck. Esta máquina de estados QuickCheck es equivalente

a la máquina de estados escrita manualmente para probar este componente en la sección 6.4.2.2 y, de hecho, las comprobaciones que se realizan en ambas máquinas de estados son exactamente las mismas, pues eso es lo que se ha descrito en la especificación OCL.

Sin embargo, incluso aunque ambas especificaciones realicen las mismas comprobaciones, no se están realizando las mismas pruebas con una aproximación que con otra, puesto que existe un punto importante que diferencia ambas especificaciones: los generadores de datos. Así, mientras que cuando la máquina de estados se escribe manualmente se han definido generadores de datos a medida que generan valores con una cierta probabilidad, o valores que cumplen unas determinadas condiciones, en la máquina de estados generada, los generadores de datos producirán valores complemente aleatorios del tipo de dato especificado en el diagrama de clases UML. Por ejemplo, el nuevo generador de `assets` es el siguiente:

```
gen_asset() ->
 {asset, [{assetId, ocl_gen:gen_string()},
       {title, ocl_gen:gen_string()},
       {summary, ocl_gen:gen_string()},
       {creationDate, ocl_gen:gen_unlimitednatural()},
       {runtime, {runtime, [
        {hours, ocl_gen:gen_unlimitednatural()},
        {minutes, ocl_gen:gen_unlimitednatural()},
        {seconds, ocl_gen:gen_unlimitednatural()}
       ]}}
 ]}.
```

Si se compara este generador con el mostrado en la página 142, se puede observar como ahora se usan generadores más genéricos. Así, mientras que antes se usaba un generador específico para generar identificadores (`gen_valid_id`), ahora se usa el generador genérico de cadenas de caracteres (`gen_string`); y de la misma forma sucede con los parámetros `creationDate` o `runtime`. Así mismo, las mejoras realizadas en la función `command` para que las operaciones se realicen sobre `assets` ya creados o nuevos `assets` no estarían presentes en la especificación de pruebas generada automáticamente.

En cualquier caso, los casos de prueba generados por la máquina de estados QuickCheck son ejecutados de la misma forma que en la aproximación en la que la especificación de pruebas se escribe manualmente, es decir, usando un adaptador que invoca las operaciones reales de la API de integración. En este caso, el adaptador es ligeramente diferente al anterior puesto que el formato del campo `runtime` no es el mismo que el usado anteriormente.

6.5.6. Resultados de aplicar la metodología de pruebas

La aproximación de pruebas descrita en esta sección se puede combinar con la metodología propuesta en la sección 6.4 para probar APIs de integración. De esta forma, la especificación de requisitos se realiza utilizando un modelo UML complementado con restricciones OCL, el cual, a parte de actuar como especificación formal de requisitos, se usará para generar los casos de prueba abstractos que posteriormente se convertirán en casos de prueba concretos a través de un adaptador.

La generación de casos de prueba abstractos se realiza a partir de propiedades o máquinas de estados QuickCheck, que, a su vez, se generan automáticamente a partir de la especificación de entrada UML y OCL, liberando, por tanto, a los probadores de software de escribir código Erlang. En este punto se ha analizado cómo actuar para generar esta especificación QuickCheck a partir del modelo UML y OCL, diferenciando entre componentes sin estado (sección 6.5.2) y componentes con estado (sección 6.5.3), los cuales a su vez podrían caer en la categoría de componentes parcialmente especificados (sección 6.5.4). Así, mientras que para los primeros se generan propiedades independientes, para los componentes con estado se generan máquinas de estados QuickCheck.

De esta forma, la tarea de escribir especificaciones QuickCheck se substituye por la definición de una especificación UML y OCL. Esto representa una ventaja en cuanto al número de líneas de código a escribir para probar un sistema software. Así, por ejemplo, en el caso de estudio del gestor de contenidos VoDKA Asset Manager, la máquina de estados QuickCheck escrita manualmente, y presentada en la sección 6.4.2 tiene un tamaño de 473 LOC, mientras que la especificación OCL, necesaria en el segundo caso, tiene 250 LOC. Esto supone una reducción de LOC del 47 %.

En algunas situaciones, reemplazar el uso de QuickCheck por especificaciones UML y OCL puede suponer un gran inconveniente en vez de una ventaja debido al uso del propio lenguaje OCL. Por ejemplo, el equipo de desarrollo de la empresa Interoud Innovation usa habitualmente el lenguaje de programación Erlang y QuickCheck, en cambio, el conocimiento del lenguaje OCL es bastante limitado. Es por ello que en este tipo de casos, la reducción de líneas de código no suele ser suficiente para preferir esta aproximación en vez de la descrita en la sección 6.4. En otras situaciones, en las que existe un equipo especializado de probadores de software, éste podría no ser un inconveniente.

Por otra parte, uno de los principales problemas de esta aproximación es la pérdida de control de los datos generados por los generadores QuickCheck. La versión actual del generador de especificaciones QuickCheck genera datos completamente aleatorios, lo cual podría no ser del todo útil en las pruebas. Así, en el ejemplo del planificador de tareas, el generador de procesos siempre generará un proceso nuevo, con un número entero aleatorio, el cual es poco probable que ya haya sido genera-

do anteriormente. Este hecho puede provocar una menor cobertura en las pruebas, puesto que es posible que no se tengan en cuenta datos que, si se implementasen las propiedades o la máquina de estados QuickCheck manualmente, sí se incluirían en las propiedades.

En cualquier caso, esto representa una mejora futura que podría realizarse. Una posible solución sería tener en cuenta las precondiciones para generar datos. Por ejemplo, la precondición de la operación `ready` del planificador de procesos indica que sólo será ejecutada si recibe un proceso marcado como `waiting`. Como el generador de procesos, `gen_process`, produce solamente procesos con identificadores aleatorios, la operación `ready` será ejecutada muy pocas veces, en concreto, únicamente cuando el proceso aleatorio generado sea un proceso ya existente y perteneciente al conjunto de procesos `waiting`. Si se tiene el cuenta que en las precondiciones se usa el operador `includes`, el generador de datos podría balancear la creación de datos incluidos en dicho conjunto, así como los no incluidos, combinando, de esta forma, pruebas positivas y negativas.

Otra posible mejora es añadir la posibilidad de incluir valores concretos para los parámetros entrada o usar generadores propios en caso de que los generadores proporcionados resultasen poco óptimos para realizar las pruebas. Esto podría realizarse a través de un módulo con *callbacks* que se llamasen desde el propio módulo de pruebas generado, comprobando si han definido una serie de *callbacks* específicos o se deben usar las funciones generadoras por defecto.

Evidentemente, la inclusión de preprocesados adicionales como el mostrado para la postcondición de la operación `init`, en el que se transforma la comprobación de que la unión de los procesos `ready` y `waiting` sea el conjunto vacío, por dos comprobaciones: que tanto `ready` como `waiting` sean conjuntos vacíos, representa una mejora en la generación de máquinas de estados QuickCheck.

6.6. Resumen

Las APIs de integración son una parte fundamental en el desarrollo de software. Es por ello que la realización de las pruebas de este tipo de APIs resulta imprescindible para conseguir un software fiable y robusto. En este capítulo se ha presentado una aproximación de caja negra que permite, a partir de una especificación abstracta de la API de integración, es decir, independiente del lenguaje de programación utilizado para implementar dicha API, generar y ejecutar casos de prueba que permiten comprobar si las implementaciones de tal API de integración cumplen su especificación funcional.

La aproximación seguida está basada en el uso de propiedades para permitir generar casos de prueba de forma automática a partir de las mismas. Además, también se ha presentado cómo la aproximación propuesta puede combinarse con las técnicas de pruebas basadas en modelos. De esta forma, los probadores de software no

necesitan conocer los detalles de un lenguaje de programación (en este caso, Erlang) para escribir las especificaciones de pruebas, sino que éstas son generadas automáticamente a partir de un modelo UML complementado con restricciones OCL, el cual es un lenguaje de modelado estándar y conocido en el ámbito informático. En concreto, se generan propiedades independientes cuando el modelo describe un componente sin estado, y máquinas de estados QuickCheck para componentes con estado.

Esta aproximación se ha ilustrado usando varios ejemplos, así como un caso de estudio real que ha permitido revelar el potencial de esta técnica. Así, ha sido posible probar varias implementaciones de una única API de integración, cada una de ellas escrita en un lenguaje de programación diferente, y todo ello con una única especificación de pruebas. Además, usando este método, han aparecido varios defectos que han permanecido ocultos durante años, después de haber superado con éxito actividades de pruebas previas, e incluso estando instalado el componente en un entorno de producción. Obviamente, estos defectos fueron identificados y, posteriormente, corregidos.

Además del caso de estudio descrito en este capítulo, esta aproximación está siendo usada para probar otras APIs de integración de otros componentes, aprovechando en algunos casos la potencia de usar una única especificación de pruebas para probar diferentes implementaciones de una misma API de integración. En concreto, ha sido usado para probar un servicio web que permite manejar datos tanto en formato XML como JSON. En este caso, se ha usado una única especificación de pruebas, pero dos adaptadores diferentes, uno para cada posible formato.

Cabe mencionar que, tanto la estructura de la metodología propuesta, como las técnicas de pruebas desarrolladas, se basan en los siguientes aspectos:

- En primer lugar, muchos programadores consideran que las pruebas son una tarea tediosa. Gracias al uso de las pruebas basadas en propiedades es posible escribir menos código de pruebas y, a la vez, generar más casos de prueba que con una aproximación en las que los casos de prueba se especifican manualmente. Si además se combina esta aproximación con las pruebas basadas en modelos, no se necesitan escribir ni siquiera casos de pruebas ni propiedades, sino que únicamente se debe escribir una especificación de cómo funciona el sistema a probar, usando para ello un lenguaje estándar de modelado como es UML, junto con restricciones OCL.

- En segundo lugar, la carencia de herramientas y metodologías estándar de pruebas provoca que éstas se realicen de una forma diferente en función de la persona que las implemente, del lenguaje de programación usado, o incluso del sistema concreto a ser probado. La metodología propuesta es independiente del lenguaje de programación usado para implementar el sistema a probar y su aplicación es posible para probar cualquier API de integración. Además,

el marco de trabajo construido representa un punto de inicio en el desarrollo de un sistema de pruebas genérico para probar APIs de integración.

Los componentes software usados en la realización de las pruebas con esta aproximación han sido generalizados para la creación de un marco de trabajo de pruebas de APIs de integración, el cual puede ser usado junto con la metodología de pruebas propuesta. Este software de desarrollo de pruebas ofrece una serie de librerías para escribir adaptadores Java, así como las herramientas necesarias para generar especificaciones QuickCheck a partir de modelos UML con restricciones OCL. De esta forma, para probar una API de integración es posible usar muchas de las partes reutilizables que se han descrito en este capítulo con el caso de estudio usado, facilitando, de esta forma, la adopción y uso de la metodología. Estas librerías pueden obtenerse de los siguientes repositorios *github*:

- Librería para la implementación de adaptadores Java: `https://github.com/miguelafr/mbt_erlang_adapter`.
- Herramienta para generar propiedades y máquinas de estados QuickCheck a partir de una especificación UML y OCL: `https://github.com/miguelafr/umlocl2eqc`.

En resumen, con esta aproximación se tratan de mitigar algunos de los problemas que suponen serios obstáculos en la práctica habitual de la comunidad de desarrollo de software acerca de las pruebas, y así, facilitar las tareas de pruebas de APIs de integración para que éstas puedan ser realizadas con todo el rigor necesario.

7

SERVICIOS WEB

7.1. Introducción

Un servicio web se define como un sistema software diseñado para soportar una interacción máquina a máquina a través de una red de comunicaciones [354]. Así, un servicio web proporciona una interfaz de comunicación ofrecida por un componente software para permitir la interacción con otros componentes a través de una red de comunicaciones, como puede ser, por ejemplo, Internet. Por tanto, los servicios web se usan para invocar servicios o intercambiar datos entre aplicaciones y, de esta forma, aportan interoperabilidad entre diferentes aplicaciones software, independientemente de sus propiedades, tecnologías usadas o de las plataformas en las que se encuentren instaladas.

El uso de servicios web está ganando popularidad, y se han convertido en una solución muy extendida a la hora de integrar componentes en la construcción de un sistema software, o para permitir la comunicación entre un sistema y otros sistemas externos de terceros, facilitando así la comunicación entre diferentes componentes [290]. De esta forma, los servicios web proporcionan un mecanismo para acceder a las funcionalidades que ofrece un componente lo suficientemente flexible como para favorecer su reutilización en otros despliegues. Basándose en esta idea, el paradigma SOC (*Computación Orientada a Servicios*) [195] usa servicios como bloques de construcción básicos que soportan el desarrollo rápido y de bajo coste de aplicaciones distribuidas en entornos heterogéneos. Así, surge la arquitectura SOA (*Arquitectura Orientada a Servicios*) [174], la cual define una estrategia para construir aplicaciones orientadas a servicios, siendo su principal objetivo proporcionar servicios que puedan ser usados por otros servicios.

Aunque los servicios web pueden funcionar a través de diferentes protocolos, como son SMTP o FTP, lo más común es usar el protocolo HTTP sobre TCP como servicio de transporte. Con respecto a la codificación de mensajes intercambiados a través de los servicios web, cabe destacar el uso de XML [357] y JSON [153] como lenguajes principales. Además, existen diferentes protocolos que se suelen usar en la implementación de los servicios web, como son XML-RPC [238], JSON-RPC [223], o los más habituales SOAP [356], y los servicios web basados en los principios REST [181, 308]. Por otro lado, existen diferentes aproximaciones que permiten describir la interfaz pública que ofrece un servicio web, como son, por ejemplo, WSDL [353] o WADL [358] que usan el lenguaje XML para describir las operaciones proporcionadas por un servicio web, así como los tipos de datos que manejan.

Los servicios web pueden considerarse como un tipo de API de integración y, de la misma forma que se discutió en el capítulo 6, las tareas de pruebas son esenciales para comprobar el funcionamiento de la implementación de los mismos. Aunque los servicios web pueden ser probados con las técnicas de pruebas descritas en el capítulo 6, en este capítulo se desarrolla una aproximación de pruebas funcionales más específica para probar servicios web, la cual se aprovecha de la forma particular en la que éstos se pueden describir, con el objetivo de simplificar la aproximación descrita en el capítulo 6 para probar APIs de integración. En concreto, el objetivo de este capítulo es describir una metodología para probar servicios web, en la que se utiliza una aproximación de pruebas basada en propiedades que hace uso de la especificación del servicio web a probar para generar casos de prueba que permitan comprobar su comportamiento.

Para ello, inicialmente se realiza una introducción a las pruebas de servicios web en la sección 7.2. Posteriormente, en la sección 7.3, se explica cómo adaptar la metodología presentada en la sección 6.4 del capítulo 6 para probar APIs de integración (usando una aproximación basada en propiedades con máquinas de estados QuickCheck) cuando dicha API de integración es un servicio web. Además, se hace una mención especial, en la sección 7.3.3, a lo que ocurre cuando el servicio web a probar evoluciona y el código de pruebas debe ser modificado. Para ello, se utiliza un caso de estudio que permite ilustrar cómo utilizar la aproximación desarrollada. A continuación, la sección 7.4, basándose en la técnica explicada en la sección 7.3, y de la misma forma que hizo en la sección 6.5 del capítulo 6, describe cómo automatizar la construcción de modelos de prueba QuickCheck a partir de la especificación WSDL del servicio web a probar, usando para ello restricciones OCL que añaden información semántica a dicha especificación WSDL. De esta forma, se describe la arquitectura de pruebas y se ilustra su funcionamiento con un caso de estudio real. Finalmente, en la sección 7.5, se realiza un resumen acerca del uso de las técnicas propuestas para probar servicios web.

7.2. Las pruebas de servicios web

Una gran variedad de técnicas han sido propuestas para probar diferentes aspectos de un servicio web. Por ejemplo, en [273] se presenta un estudio en el que se exponen multitud de métodos de pruebas de servicios web y se listan una gran cantidad de herramientas que usan estos métodos. Pruebas de unidad, pruebas basadas en modelos, verificación formal, pruebas basadas en fallos, pruebas con particiones y pruebas basadas en contratos son los métodos de pruebas cubiertos en este trabajo. No obstante, al igual que cualquier otro tipo de API de integración, las pruebas funcionales de servicios web, que son las tratadas en este capítulo, suelen realizarse desde una perspectiva de caja negra. Así, la idea principal es la obtención de casos de prueba, bien manualmente o bien generados automáticamente, que invoquen las operaciones del servicio web y comprueben que éstas se comportan de la manera esperada.

Algunos ejemplos concretos de aproximaciones de caja negra usadas para probar servicios web son las descritas en [94], donde los autores utilizan la técnica de matriz ortogonal, el método de pares definido por [279], o la técnica de partición de [96]. También existen propuestas que construyen artefactos intermedios para ayudar en el proceso como, por ejemplo, [376], que construye un autómata finito usando BPEL, o [346], que combina UML y OCL.

En cualquier caso, los conceptos descritos en el capítulo 6 para probar APIs de integración son totalmente válidos en este caso también, con la peculiaridad de que, en este caso, las operaciones de la API de integración son operaciones ofrecidas por un servicio web, normalmente accesibles usando el protocolo HTTP. Este hecho provoca que sea posible usar lenguajes de especificación particulares para servicios web, como es el caso de WSDL, el cual se usará en la aproximación propuesta en este capítulo por ser un estándar ampliamente aceptado en la industria.

El lenguaje WSDL permite describir la interfaz pública de un servicios web, especificando la sintaxis de cada una de las operaciones que forman parte del mismo, es decir, el nombre de cada operación, sus parámetros de entrada junto con los tipos de datos asociados, el resultado que produce y cómo se invoca. Sin embargo, una de las limitaciones de usar especificaciones WSDL para generar los casos de prueba es la falta de información que especifique el comportamiento, es decir, información semántica, en tal especificación. Existen algunas alternativas que pueden proporcionar este tipo de información, como son las pruebas basadas en contratos, donde los contratos definen las condiciones que se deben cumplir para poder acceder a los servicios que proporciona un componente, así como las condiciones que se deben cumplir después de la invocación de los servicios del mismo. De esta forma, existen trabajos propuestos que extienden la especificación WSDL para incluir este tipo de contratos, así como herramientas que utilizan esta especificación extendida [205, 261].

Relacionados con el uso de WSDL, también existen trabajos como es el caso de [100], donde se presenta un marco completo para realizar pruebas a partir de una especificación WSDL. En esta solución, que también se utiliza en [116], el comportamiento del servicio web se infiere utilizando algunas heurísticas a partir de la especificación WSDL del mismo, y de cierta capacidad para diferenciar respuestas erróneas. Otro ejemplo de aproximación similar se describe en [278], donde la información semántica del servicio web se suministra en el lenguaje SWRL [2]. No obstante, en todos estos enfoques hay una manifiesta falta de automatización en diferentes momentos del proceso de prueba.

Con respecto a las pruebas basadas en propiedades, que es la técnica en la que se basa la aproximación desarrollada en este capítulo, también es posible encontrar varios trabajos relacionados. Estos trabajos [235, 375] presentan un enfoque totalmente automático para probar un servicio web, generando tipos de datos e invocaciones a operaciones de dicho servicio web a partir de una especificación WSDL del mismo. La limitación radica en que las pruebas se generan únicamente desde el nivel sintáctico, puesto que no se utiliza ninguna información semántica.

7.3. Uso de propiedades abstractas para probar servicios web

Como un servicio web es un tipo de API de integración, la aproximación desarrollada para probar servicios web se basa en la propuesta genérica para probar APIs de integración presentada en el capítulo 6. Esto es, las pruebas de servicios web serán llevadas a cabo teniendo en cuenta la metodología desarrollada en la sección 6.4 para probar APIs de integración. En este caso, la API de integración es un servicio web y, por tanto, es posible usar lenguajes de especificación particulares para describir servicios web. En concreto, se ha elegido el lenguaje WSDL por ser considerado un estándar ampliamente aceptado, y el más comúnmente utilizado para describir todo tipo de servicios web. El estándar WSDL [353] está basado en el uso de XML para describir un servicio web, definiendo las operaciones que forman parte del mismo, las entradas, salidas, tipos de datos, y los protocolos específicos para acceder al servicio.

Gracias al uso de una especificación WSDL para describir el servicio web es posible automatizar ciertas partes del proceso descrito en la sección 6.4 para probar APIs de integración, en concreto, la construcción de un esqueleto de la especificación de pruebas (sección 6.4.2.2) y los adaptadores que conectan los casos de prueba generados con el sistema a probar (sección 6.4.2.5). La automatización de estas dos actividades que forman parte de la metodología de pruebas han dado lugar a una serie de herramientas, las cuales se han agrupado en un marco de trabajo llamado WSToolkit [247]. WSToolkit está formado, por tanto, por un conjunto de herramientas que automatizan tareas relacionadas con las pruebas de servicios web. A continuación se describe cómo se ha automatizado la realización de estas dos actividades en el proceso de pruebas.

7.3.1. Especificación de pruebas

La especificación de pruebas en la técnica propuesta consiste en la construcción de una máquina de estados QuickCheck cuyas operaciones se correspondan con cada una de las operaciones del servicio web a probar. La construcción de esta máquina de estados supone además la definición de un estado que almacene información relevante durante la ejecución de las pruebas, la inicialización de dicho estado, la lista de operaciones a ejecutar, los generadores de datos que generarán los valores que se usarán en dichas ejecuciones, así como las precondiciones, postcondiciones y la forma en la se modifican los datos almacenados en el estado interno para cada operación individual.

Aunque no existe ninguna información en la especificación WSDL sobre las precondiciones y postcondiciones asociadas a cada operación y, por tanto, tampoco sobre los datos que deben ser almacenados en el estado de la máquina de estados ni como éstos evolucionan a medida que se ejecutan las diferentes operaciones, sí que se dispone de información suficiente para generar una plantilla de máquina de estados QuickCheck que contenga todas las operaciones definidas en la especificación WSDL, así como los generadores de datos que generen los valores de entrada usados para cada operación [247]. Esta tarea será llevada a cabo por el marco de trabajo WSToolkit.

Una especificación WSDL contiene una referencia a un esquema XSD [359, 360]. Este esquema describe los tipos de datos y los mensajes intercambiados en las diferentes operaciones del servicio web. Estos tipos de datos se dividen en dos categorías: simples y complejos. Los tipos de datos simples pueden ser tipos de datos primitivos (enteros, decimales, cadenas de caracteres, etc.), fechas, agregaciones de éstos (como listas y uniones), o versiones restrictivas de los mismos (como, por ejemplo, enumeraciones, cadenas de caracteres que cumplen un determinado patrón, o enteros dentro de un rango). Por su parte, los tipos de datos complejos se derivan del uso de otros tipos de datos, los cuales a su vez pueden ser simples o complejos. En muchas ocasiones, los tipos de datos complejos se crean para formar agregaciones de elementos (etiquetas `sequence`, `all` o `choice`).

WSToolkit hace uso de la aplicación Erlsom [24], que proporciona una serie de funciones para parsear (y generar) documentos XML. La aplicación Erlsom soporta varios modos de funcionamiento y, entre otras funcionalidades, permite parsear documentos XML que están asociados con un esquema XSD. Así, comprueba si el documento XML cumple dicho esquema, y transforma el documento a una estructura Erlang que se basa en los tipos de datos definidos en el esquema. La versión actual de Erlsom usada por WSToolkit ignora todas las restricciones de tipos de datos simples. Sin embargo, puesto que estas restricciones son cruciales para escribir buenos generadores de datos, se ha extendido Erlsom para que extraiga información sobre los tipos de datos simples del esquema XSD y los inserte en la representación interna generada por el propio Erlsom. Esta representación interna es también la

que se usa por parte de las herramientas ofrecidas por WSToolkit.

De esta forma, la información que se especifica en el esquema XSD asociado con la especificación WSDL del servicio web es la que se usa para generar el código correspondiente a los generadores de datos usados en las pruebas. La idea consiste en mapear los posibles tipos de datos que pueden ser definidos en el esquema XSD con generadores QuickCheck, los cuales se pueden corresponder directamente con generadores ya incluidos en el lenguaje, como, por ejemplo, `integer`, `float`, `char`, `bool`, etc., o bien nuevos generadores más complejos construidos a partir de los generadores de tipos de datos simples. Este último es el caso de los tipos de datos con restricciones, como puede ser, por ejemplo, una cadena de caracteres con una determinada longitud mínima y máxima, o una cadena de caracteres que cumple una determinada expresión regular.

Para crear generadores que produzcan valores que cumplen una determinada condición se podría usar la macro `?SUCHTHAT` proporcionada por QuickCheck. Así, `?SUCHTHAT(X, G, P)` genera valores `X` con el generador `G` tales que la condición `P` es cierta. Sin embargo, usar esta estructura para resolver las restricciones soportadas en un esquema XSD, como son `enumeration`, `length`, `maxLength`, `minLength` o `pattern`, puede conllevar que el generador QuickCheck falle rápidamente con un error indicando que no es posible encontrar un valor que cumpla la condición `P` después de `N` intentos (por defecto, `N` son 100 intentos). Por tanto, se debe ser cuidadoso a la hora de construir este tipo de generadores [249].

Así, las enumeraciones pueden expresarse con el generador proporcionado por QuickCheck `oneof`, el cual genera un valor usando un elemento elegido de una lista de generadores, o `elements`, que genera un valor de una lista de valores posibles. Por ejemplo, si el esquema especifica que una cadena de caracteres puede contener el valor `male` o `female`:

```
<simpleType name="gender">
 <restriction base="string">
  <enumeration value="male"/>
  <enumeration value="female"/>
 </restriction>
</simpleType>
```

se usaría el siguiente generador:

```
gender()->
 eqc_gen:elements(["male", "female"]).
```

Otro ejemplo es un tipo de dato que represente cadenas de caracteres con una longitud entre un mínimo y un máximo:

```
<simpleType name="password">
 <restriction base="string">
  <minLength value="5"/>
  <maxLength value="8"/>
 </restriction>
</simpleType>
```

Para ello, se ha implementado el siguiente generador:

```
string(MinLen, MaxLen) ->
 ?LET(N, eqc_gen:choose(MinLen, MaxLen),
    lists:foldl(fun(_X, Acc)->
     ?LET(C, eqc_gen:choose(32, 127), [C|Acc])
    end, [],lists:seq(1,N))).
```

donde `choose(M, N)` es un generador proporcionado por QuickCheck que genera un número en el rango entre `M` y `N`; y `?LET(Pat, G1, G2)` es una macro predefinida por QuickCheck que genera un valor usando el generador `G1`, lo asocia a `Pat`, y entonces genera un valor usando el generador `G2`, el cual puede hacer referencia al valor `Pat`.

Como se comentó en la sección 6.5.6 del capítulo 6, cuando los generadores se generan automáticamente a partir de una especificación, en muchos casos, éstos no contienen toda la información necesaria para generar los casos de prueba más adecuados en cada situación concreta. Por tanto, es muy común que el usuario modifique la implementación de estos generadores para incluir más información específica del dominio y así cubrir más casos de prueba. Un caso genérico sería, por ejemplo, comprobar si el servicio web se comporta de la manera deseada cuando las operaciones se invocan con valores que no pertenecen al tipo de dato esperado. En este caso, puesto que los generadores de datos se generan automáticamente a partir de los tipos de datos especificados en el esquema XSD, nunca se generarán datos con un tipo de dato incorrecto, por lo que este caso nunca se probará si no se modifican los generadores manualmente.

Por otro lado, además de los generadores de datos necesarios para generar los valores que se usarán en los casos de prueba, las herramientas de WSToolkit también generan el esqueleto de la máquina de estados que contiene las operaciones del servicio web a probar. Evidentemente, no es posible generar la implementación correspondiente a las precondiciones, postcondiciones o función `next_state`, puesto que esta información no está presente en la especificación WSDL. Por tanto, se añadirá una implementación por defecto para cada una de estas funciones. Así, para cada operación op con parámetros $\{P^1, P^2, \ldots, P^p\}$ se generan las siguientes implementaciones por defecto:

```
precondition(_S, {call, ?MODULE, op, [{P^1, P^2, ..., P^p}]})->
 true;

postcondition(_S, {call, ?MODULE, op, [{P^1, P^2, ..., P^p}]},
   Result)->
 Result == ok;

next_state(S, _R, {call, ?MODULE, op, [{P^1, P^2, ..., P^p}]})->
 S;
```

Por tanto, como se observa, todas las operaciones podrán ser ejecutadas en cualquier momento, puesto que las funciones `precondition` siempre devuelven `true`. Las postcondiciones, por sí mismas, tampoco comprueban ningún tipo de condición. Sin embargo, se debe recordar que la invocación de las operaciones se realizará a través de un adaptador, que además, en este caso, será generado automáticamente a partir de la especificación WSDL (como se explicará en la sección 7.3.2). Este adaptador, por sí mismo, comprobará que no se han producido errores en la petición y que la respuesta cumple el modelo especificado en la especificación WSDL. Si se produce alguno de estos errores, este adaptador devuelve una tupla `{error, Code}`, donde `Code` es un código de error y, por tanto, el parámetro `Result` definido en la postcondición contendrá dicho valor; en caso contrario se devolverá el valor de retorno correspondiente.

Por otro lado, el estado, que, por defecto, nunca cambia su valor, como se puede observar en la función `next_state` anterior, se define como una tupla vacía:

```
-record(state, {}).
```

y se inicializa de la misma manera:

```
initial_state() ->
 #state{}.
```

Obviamente, se debe completar manualmente el código de las precondiciones, indicando en qué circunstancias puede ser realmente ejecutada cada operación; las postcondiciones, añadiendo código que compruebe el funcionamiento de cada operación; y la función de cambio de estado `next_state`, indicando cómo evolucionan los datos almacenados en el estado con la ejecución de cada operación; así como los propios datos que se almacenan en el estado.

El resto de código fuente generado automáticamente de la máquina de estados no necesita ninguna modificación adicional por parte del usuario. Así, la propiedad que ejecuta la máquina de estados QuickCheck siempre sigue la misma estructura:

```
prop_state_machine() ->
 ?SETUP(fun setup/0,
  ?FORALL(Cmds, eqc_statem:commands(?MODULE),
   begin
     {H, S, Res} = eqc_statem:run_commands(?MODULE, Cmds),
     eqc_statem:pretty_commands(?MODULE, Cmds, {H, S, Res},
        Res == ok)
   end)).
```

donde las funciones `setup` y `teardown` (definidas a continuación) se encargan de comenzar y parar la ejecución de la aplicación `inets`, empleada para realizar peticiones HTTP al servicio web:

```
setup() ->
 inets:start(),
 fun teardown/0.

teardown() ->
 inets:stop().
```

Finalmente, la función `command`, que especifica todas las operaciones definidas en la especificación WSDL se genera de la siguiente manera:

```
command(S) ->
 eqc_gen:oneof([
  {call, MODULE, op_1, [gen_p_1^1(S), gen_p_1^2(S), ..., gen_p_1^pi(S)]},
  {call, MODULE, op_2, [gen_p_2^1(S), gen_p_2^2(S), ..., gen_p_2^pi(S)]},
  ⋮
  {call, MODULE, op_n, [gen_p_n^1(S), gen_p_n^2(S), ..., gen_p_n^pi(S)]}
 ]).
```

donde $gen\_p_1^j$ son los generadores producidos automáticamente a partir del esquema XSD, y los comandos op_i son las funciones envoltorio definidas en el propio módulo de pruebas que invocan al servicio web a probar a través del adaptador.

De esta forma, este esqueleto de la máquina de estados QuickCheck es un módulo Erlang que puede ser compilado y ejecutado. En ese caso, se generarán casos de prueba aleatorios que corresponderán a invocaciones de las operaciones del servicio web usando los generadores de datos generados, pero únicamente se comprobará en las postcondiciones que la invocación a cada operación no ha fallado, y que la respuesta se corresponde con el tipo de dato que se indica en la especificación WSDL/XSD.

7.3.2. Implementación de los adaptadores específicos

Uno de los puntos clave de la metodología presentada en la sección 6.4 para probar APIs de integración es la construcción de adaptadores que conecten los casos de prueba generados por QuickCheck con el sistema a probar. Al contrario que ocurría con las APIs de integración genéricas, en este caso, el sistema a probar siempre va a ser un servicio web. Generar adaptadores automáticamente para invocar las operaciones de un servicio web es posible cuando se dispone de una especificación de dicho servicio web, como es el caso cuando ya existe una especificación WSDL que describe las operaciones de dicho servicio web, y la forma de acceder a ellas. Por tanto, este proceso puede ser completamente automatizado, y así, los probadores de software no necesitan construir los adaptadores manualmente.

De hecho, existen diferentes herramientas que permiten transformar especificaciones WSDL en una librería, implementada en algún lenguaje de programación genérico, que puede ser usada para invocar las operaciones del servicio web. Por ejemplo, Apache CXF [7] o Apache Axis2 [6] son dos herramientas conocidas que permiten realizar esta tarea, generando librerías Java. En este caso particular, sería interesante disponer de una herramienta que pueda generar adaptadores Erlang a partir de una especificación WSDL, puesto que, de esta forma, sería más sencilla la integración con el módulo de pruebas QuickCheck generado, que es el que dirigirá la ejecución de los casos de prueba, esto es, donde se originarán las llamadas. Puesto que no existía ninguna herramienta que realice esta tarea, se decidió implementarla. Esta herramienta forma parte de WSToolkit [247], y además de ayudar en la creación de este marco de pruebas para servicios web, supone una nueva aportación al mundo Erlang.

El uso de la aplicación Erlsom en esta herramienta permite parsear la especificación WSDL, junto con el esquema XSD, para generar un adaptador que puede invocar las operaciones del servicio web. Este adaptador es un módulo Erlang que contiene tantas funciones como operaciones del servicio web estén definidas en la especificación WSDL. Así, cada función adaptadora recibe como parámetros los datos especificados como entradas de la operación correspondiente en el fichero WSDL, codifica dichos parámetros al formato aceptado por el servicio, envía la petición al servicio invocando la operación correspondiente de la forma indicada en el fichero WSDL (GET, PUT, POST, DELETE, etc.) y, finalmente, procesa la respuesta obtenida del servicio web asegurando que no se produce ninguna excepción, y que los datos devueltos cumplen el modelo de datos especificado en el esquema XSD asociado.

Las estructuras de datos manejadas por el módulo adaptador se obtienen a partir de los tipos de datos definidos en el esquema XSD. De esta forma, la aplicación Erlsom es capaz de parsear dicho esquema y generar registros Erlang que se corresponden con cada uno de los tipos de datos definidos. Estos registros, que se definen en un fichero de cabeceras independiente del módulo Erlang adaptador, son los que

se usarán para devolver las respuestas de cada una de las operaciones del servicio web a probar (ver figura 7.1).

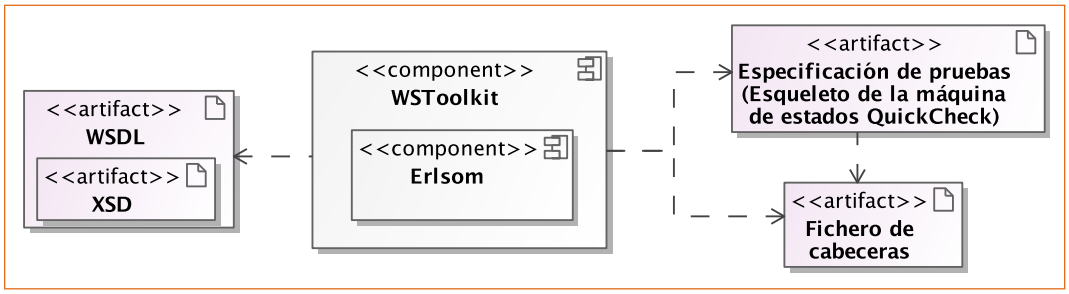


FIGURA 7.1: *Generación de máquinas de estados QuickCheck a partir de una especificación WSDL usando WSToolkit*

7.3.3. Evolución de servicios web

Los servicios web, como la gran mayoría del software en general, suelen cambiar las APIs que ofrecen con el paso del tiempo. De hecho, el proceso de diseñar una API es, en sí mismo, un proceso evolutivo, y así, los detalles de la API podrían sufrir un gran número de cambios antes de llegar a un estado estable. Un servicio web puede evolucionar de diferentes maneras, como son:

- Añadir una nueva operación al servicio web.
- Eliminar una operación del servicio web.
- Cambiar una operación ya existente del servicio web, lo cual incluye:
  - Renombrar la operación.
  - Añadir un nuevo parámetro a la operación.
  - Eliminar un parámetro de la operación.
  - Renombrar un parámetro de la operación.
  - Cambiar el tipo de dato de un parámetro de la operación.
  - Añadir más información a la respuesta devuelta por la operación.
  - Eliminar información de la respuesta devuelta por la operación.
  - Cambiar tipos de datos en la respuesta de la operación.

Los cambios de las APIs no sólo afectan a las aplicaciones cliente que las usan, sino también al código usado para probarlas. De ahí que el código de pruebas también necesite evolucionar junto con los cambios de la propia API que se prueba, lo

cual normalmente suele ser un proceso manual. Con algunas herramientas de pruebas automáticas podría ser posible regenerar el código fuente desde cero, pero, en este caso, al ser necesario que el usuario complete el esqueleto del código de pruebas generado con el comportamiento esperado del servicio web, esta aproximación no es una opción viable en la práctica.

El objetivo principal es definir una aproximación que facilite adaptar el código de pruebas cuando el servicio web que prueba ha cambiado en alguno de estos aspectos comentados anteriormente. Para ello, la herramienta clave, usada por las herramientas de WSToolkit, para soportar la evolución del código de pruebas es Wrangler [242]. Wrangler es una herramienta, implementada en Erlang, de refactorización e inspección de código Erlang que se integra con Emacs [29], XEmacs [75] y Eclipse [21].

Aparte de proporcionar un conjunto de refactorizaciones predefinidas, Wrangler proporciona una API que permite a los usuarios definir sus propias refactorizaciones o transformaciones de programas para cubrir sus propias necesidades. Esta extensibilidad es una de las diferencias clave con respecto a otras herramientas de refactorización. Para ello, se usa una técnica de análisis y transformación basada en plantillas y reglas cuyo motor se encuentra incluido dentro de la propia herramienta Wrangler [243]. Por otro lado, Wrangler también proporciona un lenguaje específico del dominio (DSL) embebido [244] que se puede usar para describir refactorizaciones compuestas a partir de refactorizaciones primitivas. De esta forma, este lenguaje proporciona un mecanismo potente y fácil de usar que permite a los usuarios programar sus propias refactorizaciones compuestas de una forma reusable para que puedan ser llevadas a cabo por lotes de una manera eficiente.

La herramienta Wrangler ya ha sido usada en algunos trabajos para la migración de APIs de integración [245], que es justamente el caso que se quiere tratar en este punto. Sin embargo, cuando lo que se quiere es evolucionar las pruebas de una API de integración que sufre cambios, codificadas como una máquina de estados QuickCheck, existen algunas particularidades que han requerido la extensión de la herramienta Wrangler para soportar el tipo de refactorizaciones necesarias, casi todas ellas debidas a la notación simbólica que se usa en la definición de las máquinas de estados QuickCheck.

Por ejemplo, si se tiene en cuenta la refactorización necesaria para añadir un nuevo parámetro de entrada a una operación ya existente, se podría pensar que los cambios a realizar en el código serían similares a la refactorización genérica *añadir un nuevo parámetro de entrada a una función*, la cual ya existe en la herramienta Wrangler. Sin embargo, éste no es el caso. Así, partiendo de una operación `op` con parámetros `P1` y `P2`, donde `gen_p1` y `gen_p2` son generadores de datos que generan valores para los parámetros `P1` y `P2` respectivamente, la máquina de estados QuickCheck contendrá el siguiente código relacionado con dicha operación `op`:

```
command(S) -> eqc_gen:oneof([
   {call, ?MODULE, op, [gen_p1(S), gen_p2(S)]},
   ...
 ]).

precondition(S, {call, ?MODULE, op, [{P1, P2}]})-> ...

postcondition(S, {call, ?MODULE, op, [{P1, P2}]}, Result)-> ...

next_state(S, R, {call, ?MODULE, op1, [{P1, P2}]})-> ...

gen_p1(S) -> ...

gen_p2(S) -> ...

op({P1, P2})-> ?ADAPTER:op(P1, P2).
```

Si se añade un nuevo parámetro a la operación `op` como primer parámetro, `P0`, además de incluir este nuevo parámetro en la función envoltorio que invoca la operación del servicio web, es necesario incluirlo en el resto de funciones del módulo de pruebas QuickCheck, las cuales hacen referencia a la operación usando la notación simbólica, así como añadir el nuevo generador de valores para el tipo de dato de este tipo de parámetros (si no existía anteriormente). Por tanto, el código de la nueva máquina de estados QuickCheck quedaría de la siguiente forma:

```
command(S) -> eqc_gen:oneof([
   {call, ?MODULE, op, [gen_p0(S), gen_p1(S), gen_p2(S)]},
   ...
 ]).

precondition(S, {call, ?MODULE, op, [{P0, P1, P2}]})-> ...

postcondition(S, {call, ?MODULE, op, [{P0, P1, P2}]}, Result)-> ...

next_state(S, R, {call, ?MODULE, op1, [{P0, P1, P2}]})-> ...

gen_p0(S) -> ...

gen_p1(S) -> ...

gen_p2(S) -> ...

op({P0, P1, P2})-> ?ADAPTER:op(P0, P1, P2).
```

La mayoría de transformaciones de este tipo se pueden llevar a cabo gracias a las plantillas y las reglas de transformación soportadas por Wrangler. Una regla de transformación define cómo se debe cambiar un programa. De esta forma, se debe reconocer un fragmento de programa a transformar, y construir un nuevo fragmento de programa que reemplace el antiguo. En Wrangler, una regla de transformación se denota por la macro `?RULE`, con el siguiente formato:

```
?RULE(Template, NewCode, Cond)
```

donde `Template` es una plantilla que representa el fragmento de código que se debe buscar para reemplazar, `NewCode` es otra expresión Erlang que devuelve el nuevo fragmento de código que lo sustituirá, y `Cond` es una expresión Erlang que se evalúa a `true` o `false` para decidir si se debe realizar la transformación o no. Todas las meta-variables/átomos declarados en `Template` son visibles en las variables `NewCode` y `Cond` y, por lo tanto, pueden ser referenciadas desde ellas. Además, es posible que `NewCode` defina sus propias meta-variables para representar fragmentos de código.

El siguiente código define una regla de transformación que añade un nuevo elemento `NewArg` en la posición `Nth` de la tupla de entrada de la operación `Op` en las referencias simbólicas a esta operación:

```
rule(Op, NewArg, Nth) ->
 ?RULE(?T("{call, M@, F@, [{Args@@}]}"),
  begin
   Args1@@ = lists:sublist(Args@@, Nth-1),
   Args2@@ = lists:nthtail(Nth-1, Args@@),
   ?TO_AST("{call, M@, F@, [{Args1@@, "++NewArg++", Args2@@}]}")
  end,
 ?PP(F@)==Op andalso api_refac:is_pattern(_This@)).
```

En este fragmento de código, las variables que terminan con el símbolo `@` son meta-variables que se corresponden con un fragmento de código (un subárbol del árbol de sintaxis abstracta del código fuente), y las variables que finalizan con `@@` son listas de meta-variables que se corresponden con una secuencia de elementos del mismo tipo. Por su parte, `This@` denota el fragmento de código completo que se encuentra con la plantilla, la cual se denota con la macro `?T`.

Aplicar esta regla al código de una máquina de estados QuickCheck añadirá un nuevo elemento a la lista de parámetros de una función en todos los lugares en los que se referencie dicha función de manera simbólica.

7.3.3.1. El proceso de refactorización del código de pruebas

Los cambios en la API proporcionada por un servicio web suponen una evolución de la especificación WSDL que describe dicho servicio web. Esta es la clave para

el funcionamiento de la herramienta de refactorización construida. Por tanto, se compararán dos versiones diferentes de una especificación WSDL, y así, de forma automática, se podrán inferir los cambios que se han realizado en la API del servicio web.

La figura 7.2 resume este proceso [247]. Así, dos especificaciones WSDL diferentes de un servicio web son parseadas y analizadas de la misma forma usando la aplicación Erlsom. Los resultados de cada análisis, presentados como un *modelo de datos y operaciones*, se usan como entrada de un *comparador* que deriva los cambios realizados tanto en los tipos de datos como en las operaciones del servicio web. El resultado de esta comparación se presenta al usuario como un *informe de cambios*. Como resultado de esta comparación entre diferentes versiones, se obtiene una lista de operaciones, cada una etiquetada con `unchanged`, `insert`, `delete`, o `substitute` en función de si no se ha modificado, es una operación nueva, la operación se ha borrado, o la operación ha sido modificada. Este resultado se procesa teniendo en cuenta las siguientes consideraciones:

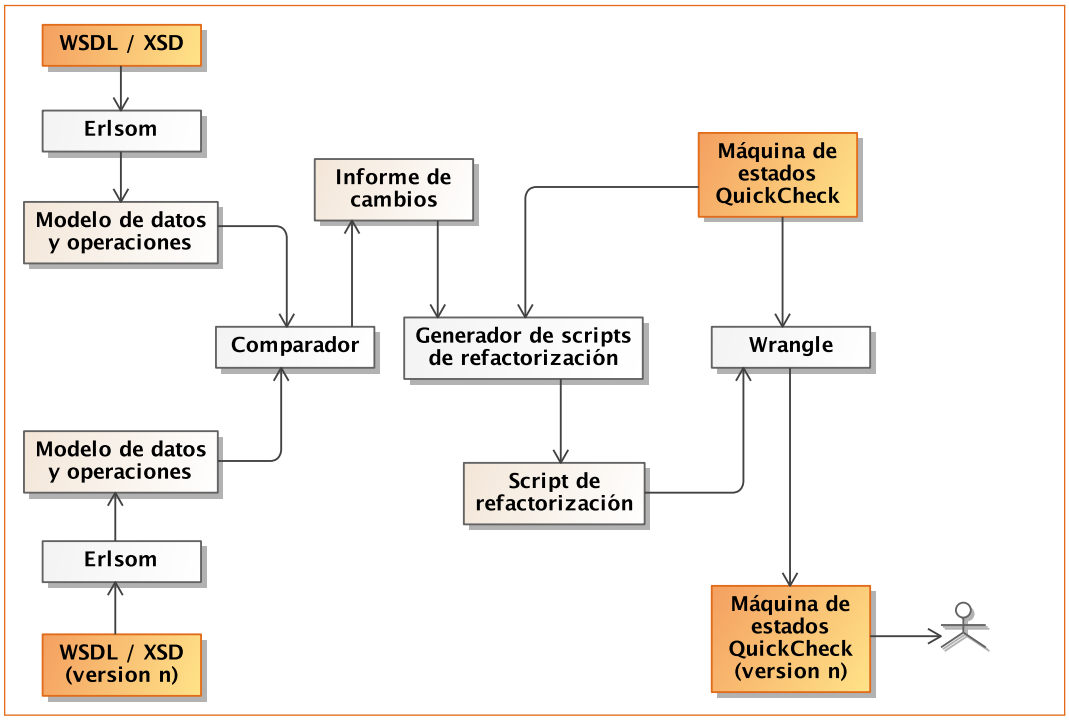


FIGURA 7.2: *Proceso de evolución del código de pruebas de servicios web*

- Se reemplaza la etiqueta `substitution` (`{substitution, A, B}`) por dos nuevas entradas `delete` (`{delete, A}`) e `insert` (`{insert B}`).
- Se eliminan las inserciones y borrados superfluos, es decir, el borrado e inserción de una misma operación. Esto ocurre, por ejemplo, cuando se reordenan

las operaciones del servicio web en la especificación WSDL, puesto que, en este caso, no es necesario que las definiciones de las funciones sigan el mismo orden que en la especificación WSDL.

- Se mezcla el borrado e inserción de dos operaciones que sólo se diferencian en su nombre en una nueva etiqueta `rename`.
- Se mezcla el borrado e inserción de dos operaciones con el mismo nombre y mismo método de acceso (GET, PUT, POST, DELETE, etc.), y se etiqueta como `input_change`, `output_change` o `input_output_change` dependiendo de dónde estén las diferencias.

Para los cambios en los parámetros de entrada y el valor de retorno, el algoritmo que se sigue para obtener el *informe de cambios* es el mismo, salvo que no se eliminan las inserciones y borrados "superfluos", ya que sí que se tiene en cuenta el orden de estos elementos (por ejemplo, los elementos definidos dentro de una etiqueta `sequence` de un tipo de dato del esquema XSD).

Este *informe de cambios* puede ser inspeccionado y modificado por el usuario, por ejemplo, en el caso en el que se hubieran realizados cambios significativos en una operación y se quiera comprobar que éstos se representan correctamente, como puede ser el renombrado y cambios en la entrada y salida al mismo tiempo. Sin embargo, lo más interesante es que este *informe de cambios* también puede ser usado por el *generador de scripts de refactorización* usando el DSL de la herramienta Wrangler. Este generador, además del *informe de cambios*, toma como entrada la implementación actual de la máquina de estados QuickCheck que prueba la versión anterior del servicio web, obteniendo como resultado un *script de refactorización* como un módulo Erlang.

El usuario puede aplicar este *script de refactorización* sobre la implementación actual de la máquina de estados a través de la herramienta Wrangler. Este proceso puede realizarse aplicando las refactorizaciones una a una, o bien aplicando todas ellas como un proceso por lotes, lo cual es deseable cuando hay un gran número de refactorizaciones a realizar. Además, es posible aplicar estas refactorizaciones de forma interactiva, de tal forma que el usuario puede tomar decisiones sobre si cada refactorización concreta debe ser realmente aplicada o no. Así, serán mostrados al usuario los cambios que debe realizar en la implementación actual de la máquina de estados QuickCheck para adaptar cada una de sus partes (función `command`, precondiciones, postcondiciones, función `next_state` y generadores) a la nueva API descrita en la nueva versión de la especificación WSDL. De esta forma, al generarse los *scripts de refactorización* automáticamente, se elimina la necesidad de que el usuario tenga que aprender cómo escribir este tipo de *scripts* manualmente. Evidentemente, el usuario podría examinar estos *scripts* y modificarlos antes de ser aplicados sobre el código fuente.

Cabe destacar que no todos los cambios realizados en la especificación WSDL

pueden ser representados como refactorizaciones. Por ejemplo, el efecto de cambiar la estructura de datos de la respuesta de una operación depende de cómo dichos datos son procesados en el modelo de pruebas, y cambiar este tipo de procesamiento para adaptarlo a la nueva estructura no es siempre posible. Por tanto, la aproximación elegida es dejar que el usuario realice dichos cambios. En este caso, la dificultad de realizar el cambio manual depende del cambio concreto que se haya realizado, aunque generalmente consiste en cambiar la implementación de la postcondición de la operación modificada (y, en ocasiones, la implementación de la función `next_state`) para tener en cuenta la nueva estructura de datos que se devuelve.

Por otro lado, con respecto al adaptador que conecta los casos de prueba con el sistema a probar, invocando directamente las operaciones del servicio web, en vez de ser refactorizado, se regenerará por completo ante una nueva implementación del servicio web, puesto que este adaptador no debe ser modificado por el usuario.

7.3.3.2. Refactorizaciones soportadas

Las refactorizaciones soportadas hasta el momento por la nueva herramienta de refactorización incluida en WSToolkit son las siguientes:

- **Añadir una operación al servicio web:** la refactorización resultante de añadir una nueva operación al servicio web consiste en añadir nuevos casos para las funciones `precondition`, `postcondition` y `next_state` correspondientes a esta nueva operación; añadir un elemento a la lista de comandos generados por la función `command`; así como la función envoltorio que invoca la nueva operación del servicio web a través del adaptador.
- **Eliminar una operación del servicio web:** como resultado de eliminar una operación del servicio web se eliminan de las funciones `precondition`, `postcondition` y `next_state` los casos correspondientes a la operación eliminada; el elemento correspondiente de la función `command` también es eliminado; así como la función envoltorio que invoca a la antigua operación haciendo uso del adaptador.
- **Añadir un parámetro de entrada a una operación del servicio web:** cuando se añade un parámetro de entrada a una operación del servicio web se modifican las funciones `precondition`, `postcondition` y `next_state` añadiendo el nuevo parámetro en la notación simbólica que referencia a la operación cambiada; también se añade el nuevo parámetro en el comando correspondiente de la función `command`; así como en la función envoltorio que invoca la operación a través del adaptador.
- **Renombrar un parámetro de entrada de una operación del servicio web:** esta es una refactorización estándar de renombrado de variable, pero necesita ser aplicada en múltiples funciones para mantener la consistencia de nombres

de los parámetros en el código de pruebas. Para ello, se ha creado una refactorización compuesta que busca en el código de prueba notaciones simbólicas que referencian a las operaciones cuyo parámetro ha sido renombrado, y realiza una refactorización de renombrado de variable para cada una de ellas.

- **Eliminar un parámetro de entrada de una operación del servicio web:** esta refactorización elimina este parámetro de las funciones envoltorio que usan el adaptador para invocar las operaciones del servicio web; de las funciones `precondition`, `postcondition` y `next_state`; y del comando especificado en la función `command`. Al eliminar parámetros de las operaciones, este tipo de refactorización puede provocar que el código resultante no compile si este parámetro estaba siendo usado en la implementación de estas funciones. Por tanto, en este caso, es necesario que el usuario lo solucione manualmente. Aunque podría eliminarse el código que referencia al parámetro eliminado para evitar problemas de compilación, se ha optado por esta opción puesto que la idea de esta aproximación es no modificar el código escrito manualmente. De esta forma, se evitan problemas que puedan surgir relacionados con borrar código que no debería ser eliminado, sino modificado.

Relacionado con el punto anterior, es importante destacar que en este proceso siempre respeta el código de la máquina de estados QuickCheck modificado manualmente por el usuario. Así, si el usuario completa la implementación de las precondiciones, postcondiciones o funciones de cambio de estado, esta implementación no se cambiará aunque la nueva implementación del servicio web lo requiera. En este caso, es el propio usuario el que debe realizar dichos cambios.

Por otro lado, algunas de las refactorizaciones pueden afectar a la definición de los generadores de datos existentes. Sin embargo, los generadores de datos generados automáticamente a partir de la información de la especificación WSDL suelen ser refinados por el usuario para generar mejores conjuntos de valores, y así, mejorar la calidad de las pruebas. Por esta razón, la aproximación seguida es no modificar el código de los generadores de datos existentes. En vez de eso, la herramienta de refactorización generará un nuevo fichero conteniendo todos los nuevos generadores de datos, es decir, generadores previamente no definidos o generadores ya definidos pero con una implementación diferente. De esta forma, será el usuario el responsable de analizar, mover y adaptar los generadores necesarios al código de pruebas. Para ello, es posible usar la propia herramienta Wrangler para aplicar una refactorización que mueva una función de un módulo a otro, o se podría realizar este proceso manualmente.

7.3.4. Caso de estudio: servicio web OSS/BSS proporcionado por VoDKATV

A continuación se ilustra cómo las técnicas explicadas en esta sección han sido usadas en las pruebas de la API de integración OSS/BSS que proporciona el componente VoDKATV-core que se muestra en la figura 3.6 de la página 56. Esta

API de integración está implementada como un servicio web, accesible mediante el protocolo HTTP o HTTPS, y maneja datos en XML. El objetivo principal de este servicio es el de ofrecer a componentes externos un mecanismo para manejar la información que gestiona el componente VoDKATV-core referente a los módulos de OSS y BSS, es decir, la provisión de usuarios, la gestión de canales, los productos y paquetes que se ofrecen a los usuarios, las listas de precios, facturación, etc. En el momento de escribir este documento, este servicio web consta de 363 operaciones de las cuales, 160 son invocadas a través del método POST y las 203 restantes se invocan mediante el método GET.

La API ofrecida por este componente es bastante extensa, por lo que, por simplicidad, únicamente se tendrá en cuenta una pequeña, pero significativa, parte de la misma. En concreto, la parte empleada para ilustrar esta aproximación se corresponde con las operaciones que manejan los *alojamientos* y los *dispositivos* en VoDKATV. Un alojamiento, conocido con el nombre de *room* en la nomenclatura de VoDKATV, representa una una unidad de facturación a la que el sistema VoDKATV dará servicio. De esta forma, un alojamiento podría ser, por ejemplo, una habitación si VoDKATV está instalado en un hotel, o una casa si VoDKATV se encarga de ofrecer sus servicios en una urbanización o bloque de apartamentos. Dentro de cada alojamiento se pueden crear uno o más dispositivos (*device*), normalmente *set-top-boxes*, los cuales se identifican a través de un identificador único, como puede ser su dirección *MAC*. Estos dispositivos permiten al usuario acceder a los servicios interactivos ofrecidos por el sistema VoDKATV. Por tanto, la parte de la API de integración considerada contiene operaciones para crear, buscar, actualizar y eliminar alojamientos y dispositivos. En total, esta parte del servicio web la componen 20 operaciones del total.

Antes de la existencia de la técnica de pruebas propuesta, en Interoud existían dos aproximaciones diferentes para probar diferentes partes de este servicio web. Por un lado, existen una serie de pruebas escritas en Java usando JUnit que se encargan de probar el funcionamiento de las operaciones del servicio web escribiendo manualmente los casos de prueba. El código fuente de las pruebas JUnit son unas 45K LOC que prueban 125 operaciones del servicio web. Por otro lado, otras operaciones del servicio web se prueban a través de una máquina de estados QuickCheck. Esta parte se corresponde con las operaciones, relacionadas con alojamientos y dispositivos, que se tendrán en cuenta en este caso de estudio. Por esta razón, está será la aproximación con la que se comparan los resultados obtenidos con la técnica propuesta. El tamaño de la máquina de estados QuickCheck existente es de 1,1K LOC, y prueba las 20 operaciones relacionadas con alojamientos y dispositivos. Además, dicha máquina de estados QuickCheck utiliza un adaptador, de 400 LOC, implementado manualmente, que se encarga de invocar las operaciones del servicio web.

Las operaciones que se tendrán en cuenta en este caso de estudio y, en general, todo el servicio web, se describen a través de una especificación WSDL. Por ejemplo,

para la operación que crea un nuevo alojamiento en el sistema, este es el fragmento de especificación WSDL que describe dicha operación:

```
<wsdl:interface name="VoDKATVInterface">
 ...
 <wsdl:operation name="CreateRoom"
    pattern="http://www.w3.org/ns/wsdl/in-out"
    style="http://www.w3.org/ns/wsdl/style/iri"
    wsdlx:safe="true">
  <wsdl:input element="msg:createRoomParams"/>
  <wsdl:output element="msg:room"/>
 </wsdl:operation>
 ...
</wsdl:interface>
...
<wsdl:binding name="VoDKATVHTTPBinding"
     type="http://www.w3.org/ns/wsdl/http"
     interface="tns:VoDKATVInterface">
 ...
 <wsdl:operation ref="tns:CreateRoom" whttp:method="POST"
    whttp:location="external/admin/configuration/CreateRoom.do"/>
 ...
</wsdl:binding>
...
<wsdl:service name="VoDKATV" interface="tns:VoDKATVInterface">
 <wsdl:endpoint name="VoDKATVHTTPEndpoint"
    binding="tns:VoDKATVHTTPBinding"
    address="http://localhost:8082/vodkatv/">
 </wsdl:endpoint>
</wsdl:service>
```

Esta especificación hace uso de los tipos de datos `createRoomParams` y `room`, definidos en el esquema XSD de la siguiente forma:

```
<wsdl:types>
 ...
 <xsd:element name="createRoomParams">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="roomId" type="xsd:string" />
    <xsd:element name="description" type="xsd:string"
       minOccurs="0" maxOccurs="1" />
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
```

```
 <xsd:element name="createRoomResponse">
  <xsd:complexType>
   <xsd:sequence>
    <xsd:element name="roomId" type="xsd:string"
       minOccurs="0" maxOccurs="1" />
    <xsd:element name="description" type="xsd:string"
       minOccurs="0" maxOccurs="1" />
    <xsd:element name="errors" type="tns:errors"
       minOccurs="0" maxOccurs="1" />
   </xsd:sequence>
  </xsd:complexType>
 </xsd:element>
 ...
 <xsd:complexType name="errors">
  <xsd:sequence>
   <xsd:element name="error" type="tns:error"
      minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="error">
  <xsd:sequence>
   <xsd:element name="code" type="xsd:string" />
   <xsd:element name="params" type="tns:errorParams"
      minOccurs="0" maxOccurs="1"/>
   <xsd:element name="description" type="xsd:string" />
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="errorParams">
  <xsd:sequence>
   <xsd:element name="param" type="tns:errorParam"
      minOccurs="1" maxOccurs="unbounded"/>
  </xsd:sequence>
 </xsd:complexType>

 <xsd:complexType name="errorParam">
  <xsd:attribute name="name" type="xsd:string" />
  <xsd:attribute name="value" type="xsd:string" />
 </xsd:complexType>

</wsdl:types>
```

Como se observa, la operación `CreateRoom` debe ser invocada mediante el método POST y recibe como cuerpo de la petición HTTP los datos de un alojamiento

en formato XML (tipo de dato `createRoomParams` definido en la especificación XSD), el cual contiene un identificador (campo `roomId`) y una descripción (campo `description`). Estos datos representan el nuevo alojamiento a crear.

A continuación se muestra un ejemplo de fichero XML con los datos de un alojamiento[1] :

```
<?xml version="1.0" encoding="UTF-8"?>
<createRoomParams>
 <roomId>ROOM_ID</roomId>
 <description>DESCRIPTION</description>
</createRoomParams>
```

Por su parte, la respuesta de dicha petición es otro fichero XML que contiene los datos del alojamiento creado, o bien un error (campo `errors`) si éste no ha podido ser creado en el sistema. En concreto, esta es la especificación del comportamiento de la operación:

- Si el identificador del alojamiento a crear (`roomId`) no se especifica, el servicio web devolverá el error `required`:

```
<?xml version="1.0" encoding="UTF-8"?>
<createRoomResponse>
 <errors>
  <error>
   <code>required</code>
   <params>
    <param value="" name="roomId"/>
   </params>
   <description>Required code</description>
  </error>
 </errors>
</createRoomResponse>
```

- Si el identificador del alojamiento a crear (`roomId`) ya existe en el sistema, el servicio web devolverá el error `duplicated`:

```
<?xml version="1.0" encoding="UTF-8"?>
<createRoomResponse>
 <errors>
  <error>
   <code>duplicated</code>
```


[1] Por simplicidad, se han eliminado algunos de los atributos asociados a los alojamientos, como son fecha de creación, una etiqueta o tipo de alojamiento.

```
    <params>
      <param value="ROOM_ID" name="roomId"/>
    </params>
    <description>Duplicated room identifier</description>
   </error>
  </errors>
</createRoomResponse>
```

- En otro caso, el alojamiento será creado, y su identificador (`roomId`) y descripción (`description`) son devueltos por el servicio web:

```
<?xml version="1.0" encoding="UTF-8"?>
<createRoomResponse>
 <roomId>ROOM_ID</roomId>
 <description>DESCRIPTION</description>
</createRoomResponse>
```

Esta especificación WSDL se ha utilizado para generar automáticamente tanto el adaptador que invocará las operaciones del servicio web como el esqueleto de una máquina de estados QuickCheck con todas las operaciones descritas en dicha especificación. Con respecto a la generación del adaptador, el código resultante es un módulo Erlang de aproximadamente 3,4K LOC que contiene una implementación para cada una de las operaciones definidas en la especificación WSDL que invocan al servicio web. Por ejemplo, esta es la implementación de la función que invoca la operación `CreateRoom` extraída del módulo adaptador generado automáticamente por WSToolkit:

```
-module(vodkatv_adapter).
...
-include("vodkatv.hrl").
...
create_room(RoomId, Description)->
 PostData = generate_post_params(createRoomParams,
    [RoomId, Description]),
 Url = ?BASE_URL++"external/admin/configuration/CreateRoom.do",
 http_request('POST', Url, PostData,
          fun(Data) ->
            process_response(createRoomResponse, Data)
          end).
```

donde `?BASE_URL` es una macro que representa la URL base definida en la especificación WSDL para acceder al servicio web, en este caso, `http://localhost:8082/vodkatv/`; `generate_post_params` es una función que transforma una serie de valores en un documento XML que cumple un determinado tipo definido en un esquema XSD, en este caso, `createRoomParams`; `http_request` es una fun-

ción que permite realizar una petición HTTP según el método especificado (GET o POST) usando para ello la aplicación Erlang `inets`; y `process_response` parsea la respuesta obtenida del servicio web y devuelve un registro Erlang según el tipo de dato definido en el esquema XSD, en este caso, `createRoomResponse`.

Además, `vodkatv.hrl` es un fichero de cabeceras con las definiciones de los registros Erlang correspondientes a los tipos de datos definidos en el esquema XSD. Así, relacionado con la creación de alojamientos, estos son los registros generados:

```
-record(createRoomParams,
      {anyAttrs :: any(),
       roomId::string(),
       description::none|string()
      }).

-record(createRoomResponse,
      {anyAttrs :: any(),
       roomId::none|string(),
       description::none|string(),
       errors::none|#errors{}
      }).

-record(errorParams,
      {anyAttrs :: any(),
       param::nonempty_list(#errorParam{})
      }).

-record(error,
      {anyAttrs :: any(),
       code::string(),
       params::none|#errorParams{},
       description::string()
      }).

-record(errors,
      {anyAttrs :: any(),
       error::nonempty_list(#error{})
      }).
```

Con respecto a la máquina de estados QuickCheck generada por WSToolkit, se ha completado manualmente la implementación de la misma, excepto las precondiciones, las cuales no se modifican puesto que cualquier operación puede ser ejecutada en cualquier momento con cualquier argumento de entrada, por lo que siempre devolverán `true` (tal y como indican las precondiciones ya generadas automáticamente):

```
precondition(_S, {call, ?MODULE, create_room,
      [_RoomId, _Description]})->
  true;
```

Así, se han definido manualmente los datos que se almacenan en el estado, los cuales, para las operaciones relacionadas con los alojamientos y los dispositivos, se componen precisamente de los datos de los alojamientos y dispositivos que se espera que se creen en el sistema VoDKATV a medida que se van ejecutando las pruebas, eliminando del estado aquellos que se espera que sean borrados durante la ejecución de las mismas:

```
-record(state, {
      rooms,
      devices }).
```

Por su parte, la función `next_state` también debe ser modificada manualmente para que se añadan o eliminen del estado los alojamientos y dispositivos en función de si se ejecuta una operación de creación o de borrado y, evidentemente, únicamente cuando el resultado de éstas se espera que sea satisfactorio. Por ejemplo, para la operación `CreateRoom` mostrada anteriormente, esta es la implementación de la función `next_state`:

```
next_state(S, _R, {call, ?MODULE, create_room,
   [RoomId, _Description]}) when RoomId =:= "" ->
  S;
next_state(S, _R, {call, ?MODULE, create_room,
   [RoomId, Description]})->
  try
    _Room = search_room(RoomId, S#state.rooms),
    S
  catch room_not_found ->
    NewRoom = #roomType{
       anyAttrs = [], roomId = RoomId, description = Description
    },
    S#state {
      rooms = [NewRoom | S#state.rooms]
    }
  end;
```

Como se observa, los alojamientos almacenados en el estado no se modificarán cuando el identificador del alojamiento a crear no se especifica (`RoomId =:= ""`), o cuando ya existe un alojamiento con el identificador del alojamiento a crear (cuando la función `search_room` no devuelve una excepción `room_not_found`). En otro caso, la operación debería ser llevada a cabo satisfactoriamente y, por tanto, se añade un nuevo alojamiento (usando el registro Erlang `roomType`) al estado.

El hecho de tener en el estado esta información almacenada permite realizar las comprobaciones oportunas en las postcondiciones para asegurar que las operaciones se comportan realmente como deben. Por ejemplo, siguiendo con la misma operación que crea un alojamiento en el sistema, ésta es la implementación de la postcondición después de ser modificado el esqueleto de la máquina de estados manualmente:

```
postcondition(_S, {call, ?MODULE, create_room,
   ["", _Description]}, Result)->
  check_simple_errors(Result#room.errors, "required",
     "roomId", "");

postcondition(S, {call, ?MODULE, create_room,
   [RoomId, Description]}, Result)->
  try
    _Room = search_room(RoomId, S#state.rooms),
    check_simple_errors(Result#room.errors, "duplicated",
       "roomId", RoomId)
  catch room_not_found ->
    Result#room.roomId == RoomId andalso
       Result#room.description == Description
  end;
```

La estructura de esta postcondición es muy similar a la estructura de la función `next_state`, es decir, se distinguen los casos en los que el identificador del alojamiento a crear es vacío (`RoomId` es `""`) o cuando ya existe un alojamiento con el identificador especificado (la función `search_room` no devuelve ninguna excepción `room_not_found`). Para realizar las comprobaciones se utiliza la función `check_simple_errors`, que comprueba si el campo `errors` de la respuesta se corresponde con el error indicado como argumento en la llamada a esta función (en este caso, `duplicated`). Por último, en otro caso, cuando el alojamiento se debería crear satisfactoriamente, se deberían devolver como respuesta los datos del alojamiento creado, por lo que debería coincidir tanto el identificador (es decir, `Result#room.roomId`) como la descripción (`Result#room.description`) con los que se pasan como parámetros (`RoomId` y `Description`).

Además de modificar las funciones `postcondition` y `next_state`, se han modificado los generadores de datos que han sido generados automáticamente por WSToolkit según el esquema XSD especificado. Así, por ejemplo, el generador de datos generado por defecto para generar identificadores de alojamientos únicamente se encargaba de generar cadenas de caracteres aleatorias con el generador `string` del módulo `gen_lib`:

```
gen_room_id(S) -> gen_lib:string().
```

Sin embargo, como se ha comentado en otras situaciones (ver sección 6.5.6 del capítulo 6) esto no garantiza que se generen identificadores que permitan probar los diferentes casos que pueden darse. Así, es interesante que el generador de identificadores de alojamientos genere valores vacíos, identificadores nuevos, e incluso identificadores de alojamientos que ya existen en el sistema. Es por ello que la implementación de este generador se cambia por esta otra:

```
gen_room_id(S)->
   gen_new_or_in_use(
    fun() -> gen_string() end,
    S#state.rooms,
    fun(Room) -> Room#roomType.roomId end).
```

donde `gen_new_or_in_use` es, a su vez, un generador de datos que recibe tres parámetros `FunGenNew`, `Used` y `FunAttrUsed`, de tal forma que la generación de valores la realiza usando el generador `FunGenNew`, o bien eligiendo aleatoriamente uno de los valores de la lista `Used` aplicándole la función del tercer parámetro `FunAttrUsed`:

```
gen_new_or_in_use(FunGenNew, Used, FunAttrUsed)->
 eqc_gen:oneof([?LET(X, eqc_gen:oneof(Used),
   FunAttrUsed(X)) || Used /= [] ++ [FunGenNew()]).
```

Además, los identificadores se componen de caracteres alfanuméricos:

```
gen_char()->
 eqc_gen:oneof([eqc_gen:choose($a, $z),
   eqc_gen:choose($A, $Z), eqc_gen:choose($0, $9)]).

gen_string() ->
   eqc_gen:list(gen_char()).
```

Después de realizar todos estos cambios manualmente, se obtiene una implementación completa de la máquina de estados QuickCheck que prueba el servicio web que proporciona VoDKATV.

Por otro lado, con este servicio web se realizaron pruebas en las que se añadían, eliminaban o modificaban operaciones, de tal forma que se ha usado WSToolkit para facilitar los cambios en la máquina de estados QuickCheck resultante. Por ejemplo, uno de los cambios realizados en la especificación del servicio web consistió en realizar las siguientes acciones:

- Añadir una nueva operación `FindAllRooms`, que se invoca a través del método GET, y devuelve todos los alojamientos existentes en el sistema.
- Añadir una nueva operación `DeleteDevice` que se invoca a través del método GET, recibe como parámetro una lista de identificadores de dispositivos

a eliminar, y devuelve la lista de identificadores de dispositivos realmente borrados.

- Modificar la operación `FindDevices`, la cual devuelve todos los dispositivos existentes en el sistema para que reciba nuevos parámetros (`sortBy`, `order` y `query`) para soportar ordenación y búsqueda en los datos devueltos. Además, también se cambian algunos tipos de datos en la respuesta para devolver información acerca de la página de resultados cuando se piden datos paginados usando los parámetros `startIndex` y `count`, los cuales, a su vez, también han sido modificados.

La realización de estos cambios produjo como resultado una nueva versión de la especificación WSDL del servicio web. Utilizando las herramientas de WSToolkit, la versión inicial de la especificación WSDL y la nueva versión que contiene estos cambios han sido comparadas. Como resultado, se obtenido el siguiente informe de cambios (al cual se le han añadido algunos comentarios manualmente para que se pueda comprender mejor):

```
[{api_added,
 {"FindAllRooms", %% nombre de la operación
  [], %% entrada
  [{room,[#room{}]}, %% respuesta
   {errors,#errors{}}],
   "GET"}}, %% método de acceso
{api_input_output_changed,
 %% operación original
 {"FindDevices",
  [{startIndex,integer()},
   {count,integer()}],
  [{device,[#device{}]},
   {errors,#errors{}}],
 "GET"},
 %% nueva operación
 {"FindDevices",
  [{startIndex,pos_integer()},
   {count,pos_integer()},
   {sortBy,none|string()},
   {order,none|orderType()},
   {'query',none|string()}],
  [{device,[#device{}]},
   {errors,#errors{}},
   {existsMore,boolean()},
   {countTotal,integer()}],
 "GET"},
```

```
 %% cambios en la entrada
 {[{input_type_changed,
    {startIndex,integer()},
    {startIndex,pos_integer()}},
  {input_type_changed,
    {count,integer()},
    {count,pos_integer()}},
  {input_added,{sortBy,none|string()}},
  {input_added,{order,none|orderType()}},
  {input_added,{'query',none|string()}}],
 %% cambios en la respuesta
 [{unchanged,{device,[#device{}]}},
  {unchanged,{errors,#errors{}}},
  {output_added,{existsMore,boolean()}},
  {output_added,{countTotal,integer()}}]}},
%% nueva operación
{api_added,
 {"DeleteDevice",
   [{deviceId,none|nonempty_list(integer())}],
   [{device,[#deletedDevice{}]},
    {errors,#errors{}}],
 "GET"}}]}
```

Usando este informe de cambios, es posible obtener automáticamente el *script* de refactorización que puede ser utilizado con la herramienta Wrangler para facilitar los cambios a realizar en el código de pruebas. Por ejemplo, el siguiente código muestra el *script* de refactorización compuesto que se obtiene automáticamente a partir del informe de cambios mostrado anteriormente:

```
-module(refac_evolve_api).

-export([composite_refac/1, input_par_prompts/0, select_focus/1]).
-include_lib("wrangler/include/wrangler.hrl").
-behaviour(gen_composite_refac).

%% callback functions.
input_par_prompts() ->[].

select_focus(_Args) -> {ok, none}.

composite_refac(_Args=#args{current_file_name=File})
 ?interactive(
   [?refac_(refac_add_op,[File,"find_all_rooms",[],[File]]),
    ?refac_(refac_add_op_arg,[File,"find_devices",1,"SortBy",
       [File]]),
```

```
?refac_(refac_add_op_arg, [File,"find_devices",1,"Order",
   [File]]),
?refac_(refac_add_op_arg,[File,"find_devices",1,"Query",
   [File]]),
?refac_(refac_add_op, [File,"delete_device",["DeviceId"],
   [File]])]).
```

Como se muestra, se han generado cinco comandos de refactorización, dos de ellos para añadir nuevas operaciones, y tres para añadir nuevos parámetros de entrada. El número usado como parámetro en los comandos `refac_add_op_arg` indica la posición donde el nuevo parámetro debe ser añadido. Por ejemplo, `1` indica que el nuevo parámetro debería ser añadido al final de la lista de parámetros, es decir, el primer elemento de la tupla que contiene los parámetros, pero comenzando a contar desde el final.

7.3.5. Resultados de aplicar la metodología de pruebas

La técnica presentada en esta sección se integra dentro de la metodología de pruebas descrita en la sección 6.4 del capítulo 6 para probar APIs de integración. Así, con una aproximación basada en propiedades, usando QuickCheck, es posible probar un servicio web de la misma forma que se propone hacerlo para una API de integración en general. Sin embargo, en este caso, aprovechando el hecho de que la API de integración es un servicio web, es posible automatizar ciertas partes del proceso de pruebas, muchas de ellas tediosas de escribir manualmente, consiguiendo que los probadores de software únicamente centren sus esfuerzos en implementar aquellos fragmentos de código que realmente comprueben el funcionamiento del sistema, mejorando, de esta forma, la eficiencia y eficacia en la realización de las pruebas.

Así, se han implementado varias herramientas que componen el marco de trabajo WSToolkit que se encargan de facilitar la tarea de realizar las pruebas de un servicio web. Estas herramientas permiten construir el esqueleto inicial de una máquina de estados QuickCheck que invoca todas y cada una de las operaciones definidas en una especificación WSDL, y donde los valores usados para cada una de estas invocaciones se generan automáticamente mediante generadores de datos que son, a su vez, generados a partir de los tipos de datos definidos en el esquema XSD.

Este esqueleto inicial de la máquina de estados QuickCheck es un modelo de pruebas compilable y ejecutable. De hecho, aunque se debe completar dicho modelo con el código que realmente comprueba el comportamiento de cada operación, así como con mejores generadores de datos si es necesario, este esqueleto inicial ya realiza por sí mismo ciertas comprobaciones sobre la ejecución de las operaciones. En concreto, el modelo inicial fallará si alguna operación no se puede invocar, o bien si la respuesta devuelta por la misma no se corresponde con el tipo de dato

definido en la especificación WSDL/XSD. Además, la máquina de estados resultante de este proceso es una representación abstracta del modelo, pues es válida para probar cualquier implementación del servicio web que se corresponda con la especificación WSDL. Esta abstracción se consigue gracias al uso del adaptador, el cual, en este caso, también es automáticamente generado por las herramientas incluidas en WSToolkit.

También se facilita la tarea de cambiar el código de pruebas a medida que el código del servicio web evoluciona. Para esto, la herramienta construida se basa en la comparación de dos versiones diferentes de una especificación WSDL, a partir de la cual se genera un informe de cambios que, a su vez, puede ser usado para generar automáticamente un *script* de refactorización para poder ser aplicado al código de pruebas con la herramienta Wrangler. Así, se evita la necesidad de examinar manualmente los cambios en las diferentes interfaces para averiguar qué modificaciones realizar en el código de pruebas. Con esta aproximación, es la propia herramienta la que analiza los cambios realizados y sugiere al usuario los cambios que debe realizar en la máquina de estados. Este proceso reduce los errores humanos que se pueden producir a la hora de actualizar manualmente el código de pruebas.

Esta aproximación ha sido usada dentro de la empresa Interoud Innovation para probar sus servicios web como parte de un piloto del proyecto europeo FP7 PROWESS. De hecho, el caso de estudio presentado en la sección 7.3.4 es parte de este piloto [88]. Una de las primeras consecuencias positivas que se ha podido extraer del uso de WSToolkit ha sido la generación automática del adaptador a partir de la especificación WSDL. Hasta la existencia de esta herramienta, cuando se usaba una especificación QuickCheck, estos adaptadores eran implementados manualmente por el personal de Interoud Innovation, puesto que, por simplicidad, se implementaban en Erlang ya que la especificación de pruebas también lo estaba, y hasta este momento no existía ninguna herramienta que realizase esta tarea automáticamente. Así, por ejemplo, la máquina de estados QuickCheck existente antes de esta aproximación hacía uso de un adaptador implementado manualmente de 400 LOC que permitía invocar 20 operaciones del servicio web. De esta forma, el ahorro de tiempo y esfuerzo con respecto a la construcción del adaptador fue considerable, evitando, además, errores humanos que pueden ocurrir en la implementación del mismo. Además, cabe mencionar que el módulo adaptador generado automáticamente, de 3,4K LOC, permite invocar las 363 operaciones descritas en la especificación WSDL.

Por otro lado, con la generación automática del esqueleto de la máquina de estados QuickCheck, si bien el código resultante no comprueba el comportamiento de cada una de las operaciones a no ser que sea complementado posteriormente de manera manual, sí que se consigue automáticamente un código de pruebas que invoca todas y cada una de las operaciones descritas en la especificación WSDL. En el caso del servicio web proporcionado por el componente VoDKATV-core, esto

ha supuesto la construcción automática de un esqueleto de una máquina de estados QuickCheck de aproximadamente 2,5K LOC que incluye la invocación de las 363 operaciones definidas en la especificación WSDL, la cual ha sido modificada manualmente para añadir unas 950 LOC que definen el comportamiento de las 20 operaciones consideradas. De hecho, gracias a esta tarea han sido incluidas en el módulo de pruebas operaciones del servicio web que proporciona VoDKATV para las que no existía ningún tipo de prueba hasta el momento, ni siquiera únicamente su invocación. Además, como en Interoud Innovation ya se estaban usando especificaciones WSDL para la definición de los servicios web, este paso no supuso ningún tipo de coste adicional.

Con respecto a la evolución de servicios web, se analizó cómo éstos evolucionaban dentro de Interoud Innovation, y se observó como en septiembre y octubre del año 2013, diez nuevas operaciones fueron añadidas al servicio web OSS/BSS, y quince operaciones existentes fueron modificadas, muchas de ellas extendiendo su funcionalidad con nuevos parámetros. Antes de la existencia de estas herramientas, el proceso de refactorización de código de pruebas en Interoud Innovation se realizaba manualmente, analizando los cambios y examinando qué partes del código de prueba se deben cambiar. Sin embargo, la posibilidad de usar Wrangler para cambiar el código de pruebas ha facilitado, en muchos casos, la realización de este tipo de tareas ya que, al utilizar una herramienta que guíe los cambios a realizar, el equipo de personas que mantiene el código de pruebas no olvida cambiar ninguna de las partes afectadas de la máquina de estados, como ocurre cuando este proceso se realiza manualmente.

En conclusión, aunque todavía es necesario modificar la máquina de estados QuickCheck manualmente después de que el esqueleto de la misma haya sido generado, la opinión general de las herramientas incluidas en WSToolkit dentro de la empresa Interoud Innovation ha sido positiva. Así, aunque la parte mejor valorada, y que realmente supone una nueva contribución al mundo Erlang, es la generación automática de adaptadores, también se valora positivamente la construcción automática de una máquina de estados QuickCheck que invoque todas las operaciones definidas de un servicio web, así como el nuevo proceso de refactorización de código de pruebas más robusto.

7.4. Generación automática de propiedades a partir de WSDL y OCL

De la misma forma que para probar APIs de integración genéricas es posible generar automáticamente un modelo de pruebas QuickCheck, como se muestra en la sección 6.5 del capítulo 6, en la que dicho código de pruebas se genera a partir de un modelo UML con restricciones OCL que describen el sistema a probar; para los servicios web también es posible seguir una aproximación similar. Así, esta sección muestra una aproximación en la que a partir de una especificación del servicio web

a probar se genera un modelo de pruebas QuickCheck que, a su vez, se usará para generar casos de prueba usando una aproximación basada en propiedades, las cuales permiten comprobar el comportamiento de dicho servicio web.

En este caso, al tratarse de un servicio web, no es necesario usar el lenguaje de modelado UML, sino que es posible utilizar lenguajes de especificación para este tipo de sistemas, como es el caso del lenguaje WSDL. Sin embargo, el lenguaje WSDL opera a nivel sintáctico y no es posible representar los requisitos o condiciones de funcionamiento del servicio web usando únicamente la información proporcionada con esta especificación. Haciendo una analogía con el uso de modelos UML con restricciones OCL para probar APIs de integración, la especificación WSDL se correspondería con el modelo UML, puesto que únicamente describe información sintáctica.

Por tanto, para poder usar una aproximación similar a la descrita para APIs de integración es necesario añadir información semántica a la descripción del servicio web, es decir, la descripción WSDL debe completarse con anotaciones semánticas que expresen precondiciones y postcondiciones para cada operación del servicio web. Actualmente, se han propuesto varias opciones para llevar a cabo esta tarea, como es el uso de WSDL-S [355], SWRL [2], u OCL [129, 363]. El concepto de añadir este tipo de información a la especificación de un servicio web produce como resultado los *servicios web semánticos*, los cuales han sido enriquecidos con una descripción semántica de su comportamiento. Este tipo de descripciones semánticas son expresadas con *ontologías*, una conceptualización formal de un dominio particular. Así, la descripción semántica se crea usando componentes del dominio y expresando las relaciones entre ellos.

En este caso concreto, debido a la experiencia previa en el uso de OCL como lenguaje de especificación semántica de operaciones de APIs de integración, también se ha elegido esta opción para complementar la especificación WSDL. Para ello, se usa el estándar WSDL-S para añadir, a través del atributo `modelReference`, una relación entre la especificación WSDL y la información semántica expresada en el lenguaje OCL. Así, usando el lenguaje OCL, se describen las precondiciones y postcondiciones de las operaciones incluidas en la especificación WSDL del servicio web.

Por tanto, así como en APIs de integración genéricas se usaba un modelo UML con restricciones OCL, para servicios web se usará una especificación WSDL también acompañada de restricciones OCL. A partir de esta especificación de entrada se generará un modelo QuickCheck que permitirá a su vez generar, usando una aproximación basada en propiedades, casos de prueba que permitan comprobar, desde una perspectiva de caja negra, el comportamiento de las operaciones descritas en la especificación WSDL y extendido con las restricciones OCL [190, 250, 253].

7.4.1. Arquitectura de pruebas

La figura 7.3 muestra los componentes que forman parte de la arquitectura de pruebas propuesta. Como se observa, el diagrama es muy similar al que se muestra en la figura 6.12 de la página 166, el cual representaba los componentes necesarios para probar una API de integración en general. En este caso, la diferencia principal radica en el hecho de que no se usará un modelo UML como entrada al generador de ATS, sino que se usará la especificación WSDL del servicio web a probar (la cual también se complementará con restricciones OCL).

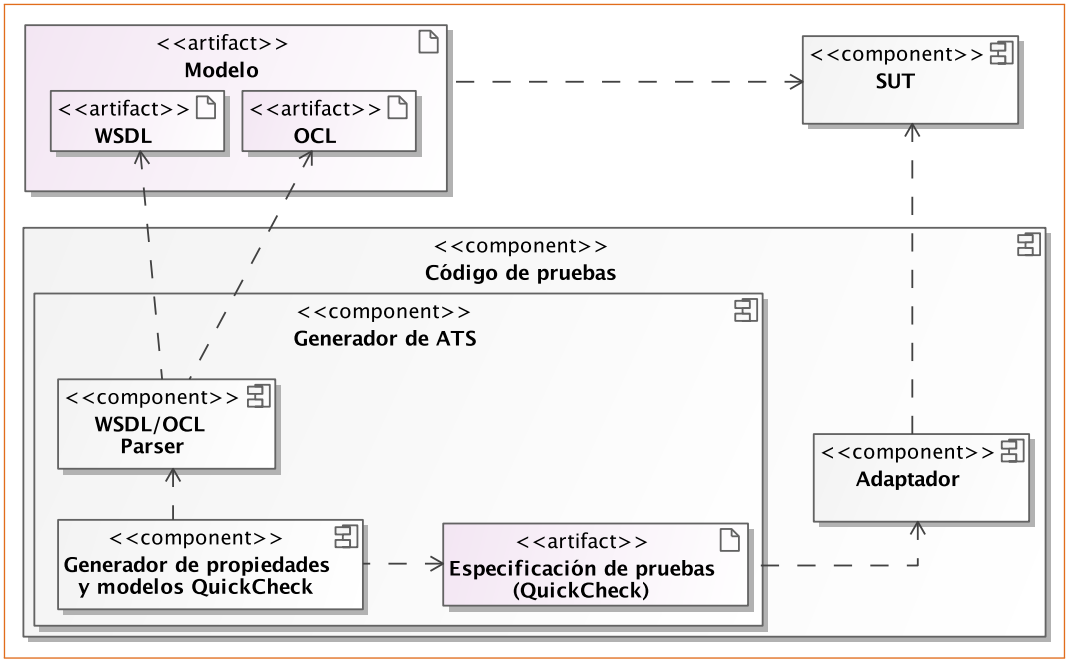


FIGURA 7.3: *Arquitectura propuesta para probar servicios web*

Para ello, se ha construido un *parser* de ficheros WSDL, implementado en Haskell y que usa la librería `Text.XML.Light`. Este componente es capaz de leer la especificación WSDL de un servicio web, y extraer toda la información necesaria de dicha especificación para que el componente encargado de generar la especificación de pruebas QuickCheck sea capaz de generar dicho código de pruebas.

Por ejemplo, el siguiente extracto de especificación WSDL describe un servicio web que proporciona la operación `pow`, la cual puede ser invocada a través del método GET. Esta operación recibe dos parámetros, un entero `a` y un entero positivo `b`, y devuelve otro entero como resultado:

```
<wsdl:types>
 <xs:schema ...>
  <xs:element name="powParams">
   <xs:complexType>
    <xs:sequence>
     <xs:element minOcurs="0" name="a" type="xs:int" />
     <xs:element minOcurs="0" name="b"
             type="xs:positiveInteger" />
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  <xs:element name="powResponse">
   <xs:complexType>
    <xs:sequence>
     <xs:element minOccurs="0" name="return"
             type="xs:int"/>
    </xs:sequence>
   </xs:complexType>
  </xs:element>
  ...
 </xs:schema>
</wsdl:types>

<wsdl:interface name="MathUtilsInterface">
 <wsdl:operation name="pow"...>
  <wsdl:input element="msg:powParams"/>
  <wsdl:output element="msg:powResponse"/>
  <wssem:modelReference="powConstraints"/>
 </wsdl:operation>
</wsdl:interface>

<wsdl:binding name="MathUtilsHTTPBinding"
 type="http://www.w3.org/ns/wsdl/http"
 interface="tns:MathUtilsInterface">
 <wsdl:operation ref="tns:pow" whttp:method="GET"
  whttp:location="pow"/>
</wsdl:binding>

<wsdl:service name="MathUtils"
         interface="tns:MathUtilsInterface">
 <wsdl:endpoint name="MathUtilsHTTPEndpoint"
  binding="tns:MathUtilsHTTPBinding"
  address="http://localhost:8080/mathUtils/">
 </wsdl:endpoint>
</wsdl:service>
```

Para esta especificación WSDL, el *parser* necesita extraer la siguiente información:

- El nombre del servicio web a partir de la etiqueta `service`, en este caso, `MathUtils`.
- El nombre de cada operación a partir de las etiquetas `operation` que se encuentran dentro del elemento `interface`. En este caso, la única operación definida es `pow`.
- Las entradas y salidas de cada operación, a través de las etiquetas `input` y `output` correspondientes a cada operación definida, es decir, `powParams` y `powResponse` para la operación `pow`.
- Los tipos de datos referenciados en las entradas y salidas (etiqueta `types`). En el ejemplo, para la operación `pow`, se definen, en el esquema XSD, los tipos de datos `powParams` y `powResponse`.
- La URL para invocar cada operación del servicio web, la cual se construye a partir de la URL especificada en el atributo `address` de la etiqueta `endpoint` y el atributo `location` para cada operación definida dentro del elemento `binding`. Así, la operación `pow` se invoca realizando una petición GET a la URL `http://localhost:8080/mathUtils/pow`.
- El atributo `modelReference` de WSDL-S que apunta al fichero externo OCL, es decir, `powConstraints` en el ejemplo.

Este fichero OCL al que se hace referencia debe ser, por tanto, parseado, para comprobar si existe información semántica asociada a las operaciones referenciadas. Puesto que todas las herramientas existentes que parsean ficheros OCL requieren un modelo UML para su funcionamiento, como es el caso de Dresden OCL [159, 160], usado en este trabajo para este fin, se ha implementado un *parser* propio de OCL. Para ello, se ha aprovechado parcialmente el trabajo realizado por el proyecto OCLNL [49]: una gramática BNF etiquetada [183] para OCL. Esta gramática se utiliza con una herramienta de generación de compiladores (BNFC [13]) para construir el árbol de sintaxis abstracta, el analizador léxico y el *parser*.

Por ejemplo, para la especificación WSDL mostrada anteriormente, el contenido del fichero `powConstraints` podría ser el siguiente:

```
context MathUtils::pow(a: Integer,
                b: UnlimitedNatural): PowResponse
post post_pow: result.return = Sequence{1..b}->
  iterate(i : Integer; acc : Integer = 1 | acc*a)
```

el cual especifica, utilizando el lenguaje OCL, cómo se debe comportar la operación `pow`.

Finalmente, una vez que se obtiene la información requerida, es posible generar el modelo de pruebas QuickCheck. Para ello, se reaprovecha el trabajo realizado para generar el código de pruebas en APIs de integración, en concreto, el componente que genera las propiedades y modelos QuickCheck, descrito en la sección 6.5 del capítulo 6. La forma de conseguir esto es que los *parsers* de WSDL y OCL construidos generan la misma representación interna con la que el sistema ya trabajaba a partir de un modelo UML con restricciones OCL.

A partir de este punto, de la misma forma que se explicó para las pruebas de APIs de integración en general, dependiendo de los requisitos del servicio web, el modelo de pruebas generado estará formado por una serie de propiedades independientes para servicios web sin estado, o una máquina de estados para servicios web con estado. En cualquier caso, ya sea a partir de propiedades o a partir de una máquina de estados, QuickCheck generará casos de prueba que podrán ser ejecutados. Para ello, se necesita la ayuda de un adaptador que conecte dichos casos de prueba con el sistema a probar. No obstante, este adaptador se generará automáticamente a partir de la especificación WSDL usando las herramientas contenidas en el marco de trabajo WSToolkit, como se ha explicado en secciones anteriores. Por tanto, al generar el modelo de pruebas QuickCheck se debe tener en cuenta cómo va a ser el formato de las respuestas devuelto por tal adaptador.

7.4.1.1. Servicios web sin estado

Los servicios web sin estado son aquellos ofrecidos por componentes que no tienen un estado interno que afecte al resultado de invocar una operación. Por tanto, la respuesta de una operación depende únicamente de los argumentos que recibe, y no de otras operaciones que hayan podido ser ejecutadas anteriormente. Para este tipo de servicios web, la aproximación seguida es la misma que la que se explicó en la sección 6.5.2 para probar APIs de integración de componentes sin estado, con la diferencia de que ahora la lista de operaciones a probar, junto con sus parámetros de entrada y valor de retorno, están descritas en una especificación WSDL en vez de un modelo UML.

En cualquier caso, como se ha implementado un proceso de traducción que transforma la especificación WSDL/OCL a la misma representación interna a la que se transforma el modelo UML/OCL con la que trabaja el componente generador de propiedades y modelos, el proceso que se sigue después de dicha transformación es exactamente el mismo en ambos casos. De esta forma, usando la especificación WSDL para obtener la lista de operaciones a probar junto con sus parámetros de entrada y valor de retorno, así como las restricciones OCL asociadas, se generan una serie de propiedades que comprueban si la ejecución de cada operación cumple las postcondiciones asociadas definidas con dichas restricciones OCL. Así, la propiedad generada que prueba que una operación op que recibe p parámetros $P^1, \ldots, P^p$,

y que lleva asociadas una serie de precondiciones pre, y cumple un conjunto de postcondiciones $post_1, post_2, \ldots, post_M$, es la siguiente:

```
prop_op() ->
  ?FORALL(P, G(), ?IMPLIES(pre(P),
   begin
     R = op(P),
     post_1(P, R) andalso
        post_2(P, R) andalso
        ⋮
        post_M(P, R)
   end)).
```

la cual es usada por QuickCheck para generar valores aleatorios que cumplan las precondiciones, con los cuales se ejecutará la función a probar (usando el adaptador HTTP generado automáticamente por las herramientas de WSToolkit), y comprueba si la postcondición especificada se cumple.

Por ejemplo, el servicio web `MathUtils` mostrado en la sección anterior es un servicio web sin estado, puesto que en la especificación OCL asociada a sus operaciones no se usa el operador `@pre`. Por tanto, se generará una propiedad como la mostrada anteriormente para cada una de las operaciones que contiene. Así, teniendo en cuenta las restricciones OCL mostradas para este ejemplo, la propiedad que prueba la función `pow` es la siguiente:

```
prop_pow() ->
  ?FORALL({A, B}, {ocl_gen:int(), ocl_gen:nat()},
   begin
     Result = mathUtils:pow(A, B),
     ocl:tag([
       {"post_pow",
          ocl_datatypes:get_property(return, Result) ==
             ocl_seq:iterate(
               fun (I, Acc) ->
                 Acc * A
               end, 1, ocl_seq:new(1, B))
       }
     ])
   end).
```

Con esta especificación de pruebas, la herramienta QuickCheck se encargará de generar aleatoriamente números enteros (usando el generador `int` del módulo `ocl_gen`) y números naturales (usando el generador `nat` del módulo `ocl_gen`) que se usarán como datos de entrada en la ejecución de la función a probar `pow` del módulo `mathUtils`. En este caso, cuando se ejecuta esta propiedad, la herramienta

QuickCheck devuelve el siguiente error:

```
> Testing property: prop_pow
................................Failed! Reason:
{'EXIT',{bad_property,"post_pow"}}
After 52 tests.
{-13,16}
Shrinking..(2 times)
Reason:
{'EXIT',{bad_property,"post_pow"}}
{13,15}
```

que significa que la ejecución de las pruebas ha revelado una inconsistencia entre la propiedad especificada y la implementación del servicio web. En este caso particular, el error es un conocido error de redondeo con números flotantes, como se explica en [87].

7.4.1.2. Servicios web con estado

Al contrario que los servicios web sin estado, donde cada operación es independiente del resto, en los servicios web con estado el resultado de una operación puede depender de qué operaciones hayan sido invocadas anteriormente, además de los argumentos de entrada recibidos. Este es el caso más común en los servicios web reales. En este caso, propiedades independientes que prueban las operaciones proporcionadas por este servicio web no son útiles, porque el orden en el que éstas son ejecutadas pueden modificar el resultado de las pruebas, es decir, las operaciones del servicio web causan efectos colaterales que afectan a su comportamiento consiguiente.

Por tanto, al igual que en las pruebas de APIs de integración de componentes con estado, el estado interno se tiene que tener en cuenta para el proceso de pruebas. Así, el código de pruebas resultante será una máquina de estados QuickCheck que generará secuencias de llamadas a las operaciones definidas en la especificación WSDL del servicio web, comprobando que las precondiciones y postcondiciones definidas en el fichero OCL asociado se cumplen. En cualquier caso, gracias a la reutilización del componente generador de propiedades y modelos, el proceso que se sigue es exactamente el mismo que el que se explica en la sección 6.5.3 para APIs de integración.

7.4.2. Caso de estudio: API de gestión de VoDKATV

Para ilustrar el funcionamiento de esta técnica se tomó el mismo ejemplo usado en la sección 7.3.4, es decir, el servicio web que ofrece el componente VoDKATV-core para manejar la información relacionada con el subsistema de OSS/BSS. Evidentemente, este es un ejemplo de servicio web con estado, puesto que la creación

de nuevos elementos, como son los alojamientos y los dispositivos, provocarán que haya operaciones que varíen su comportamiento al ser ejecutadas. Por ejemplo, eliminar un alojamiento que no existe devolverá un error, pero si esta operación se invoca después de crearlo, entonces el alojamiento se eliminará del sistema.

De la misma forma que se hizo anteriormente, se ilustra cómo se debería expresar el comportamiento de la operación `CreateRoom`, pero esta vez usando restricciones OCL en vez de escribir código Erlang directamente. Se debe recordar que la operación `CreateRoom` crea un nuevo alojamiento en el sistema VoDKATV siempre y cuando el alojamiento a crear tenga un identificador no vacío, y no exista otro alojamiento con el identificador especificado. En caso contrario, se devolverá un error `required` o `duplicated` respectivamente.

Para este caso, es necesario definir en la especificación OCL una variable auxiliar, que actuará como variable de estado, para almacenar los alojamientos que se esperan que sean creados en el sistema VoDKATV a medida que se ejecutan las pruebas:

```
context VoDKATVInterface

def: state_rooms:Set(Tuple(roomId:String, description:String)) =
   Set{}
```

Esta variable se utilizará para escribir la postcondición de la operación considerada `CreateRoom`. Además, las entidades `complexType` y `element` definidas en la especificación WSDL/XSD pueden ser accedidas como clases UML desde el código OCL, donde los tipos de datos definidos en el esquema XSD serán considerados como clases UML, mientras que los elementos de los mismos podrán ser accedidos como si fueran atributos de estas clases. Con esta aproximación, la especificación para la operación `CreateRoom` puede ser escrita en OCL mediante el siguiente código:

```
context VoDKATVInterface::CreateRoom(
   roomId:String, description:String): CreateRoomResponse
 post CreateRoom:
   if ((roomId = "") or (roomId = null)) then
     (self.state_rooms = self.state_rooms@pre and
        result.errors->size() = 1 and
        result.errors->at(0).code = "required" and
        result.errors->at(0).params->size() = 1 and
        result.errors->at(0).params->at(0).name = "roomId" and
        result.errors->at(0).params->at(0).value = "")
   else
```

```
    if (self.state_rooms->select(
       room | room.roomId = roomId)->notEmpty()) then
      (self.state_rooms = self.state_rooms@pre and
        result.errors->size() = 1 and
        result.errors->at(0).code = "duplicated" and
        result.errors->at(0).params->size() = 1 and
        result.errors->at(0).params->at(0).name = "roomId" and
        result.errors->at(0).params->at(0).value = roomId)
    else
      result.roomId = roomId and
        result.description = description and
        self.state_rooms = self.state_rooms@pre->including(
         Tuple {
          roomId:String = roomId,
          description:String = description
         })
    endif
   endif
```

Así, usando el generador de propiedades y modelos QuickCheck, esta especificación, la cual, como se observa, usa el operador `@pre` de OCL, se usará para generar una máquina de estados QuickCheck con todas las operaciones listadas en la especificación WSDL del servicio web, entre ellas, `CreateRoom`, y definiendo para cada una ellas sus precondiciones, postcondiciones y funciones de cambio de estado según la información especificada en el fichero OCL. Por ejemplo, este es un fragmento de la postcondición generada para la operación `CreateRoom`, en concreto, donde se comprueba si el identificador del alojamiento a crear es una cadena de caracteres vacía:

```
postcondition(PreState, AfterState, {call, ?MODULE, createRoom,
   [RoomId, Description]}, R) ->
  case RoomId of
    "" ->
     Result = ocl_datatypes:get_property(errors, R),
     ocl_set:eq(AfterState#ts.rooms, PreState#ts.rooms) andalso
       ocl_string:eq(ocl_datatypes:get_property(code, R))),
         "required")
   ...
```

La máquina de estados QuickCheck resultante usará el adaptador HTTP generado automáticamente para invocar las operaciones del servicio web. Para ello, cada una de las operaciones serán invocadas con valores aleatorios generados a partir de generadores de datos que son, a su vez, generados automáticamente a partir de los tipos de datos definidos en el esquema XSD.

7.4.3. Resultados de aplicar la metodología de pruebas

La generación automática de código de pruebas QuickCheck a partir de una especificación WSDL y OCL ha sido evaluada dentro del proyecto europeo FP7 PROWESS [4] a través de un piloto de estudio [88], en concreto, el mismo ejemplo que se explicó en la sección anterior, es decir, el servicio web que proporciona el sistema VoDKATV para manejar la información relacionada con el subsistema OSS/BSS. Como parte de este estudio, se han codificado en OCL las precondiciones y postcondiciones de todas las operaciones relacionadas con alojamientos y dispositivos, entre las que se encuentra la operación que crea un nuevo alojamiento descrita en la sección anterior. En total, se ha incluido la especificación OCL de 20 operaciones del servicio web. De esta forma, se ha conseguido como resultado un fichero OCL de 519 LOC.

Teniendo en cuenta que el código del adaptador HTTP se genera automáticamente y no debe ser escrito manualmente, si se comparan las líneas de código que se deben escribir para codificar la máquina de estados QuickCheck manualmente con las necesarias para que dicha máquina de estados se genere automáticamente, se puede observar como se ha obtenido una reducción del 52 % de código, pasando de necesitar escribir 1,1K LOC de código Erlang a 519 LOC de código OCL.

Otro punto positivo de esta aproximación es que la ejecución de las pruebas usando la máquina de estados QuickCheck generada automáticamente a partir de la especificación WSDL y OCL reveló tres defectos en el sistema VoDKATV:

- Un problema con la codificación de caracteres en ficheros XML enviados por POST al servicio web cuando contienen algunos caracteres como ê, Ò, etc. Tales peticiones provocan que el servidor devuelva un error desconocido (código de estado 500 del protocolo HTTP), y, por tanto, la operación no sea ejecutada de la manera esperada. Estos caracteres no se tuvieron en cuenta en los generadores QuickCheck que han sido escritos a mano, puesto que, en ese caso, el generador de cadenas de caracteres únicamente generaba caracteres alfanuméricos. Sin embargo, estos caracteres sí que son generados por esta aproximación, puesto que cadenas de caracteres con estos caracteres cumplen la especificación WSDL. Este defecto fue solucionado en la implementación del servicio web.

- Un problema en varias peticiones cuando se usaban índices negativos en los parámetros utilizados para la iteración por páginas de los resultados (parámetros `startIndex` y `count`). De nuevo, estas peticiones causaban un error no controlado. Sin embargo, en este caso, este fallo fue considerado un defecto en la especificación WSDL, puesto que no debería permitir valores negativos en estos parámetros. Así, se modificó la declaración de estos parámetros cambiando el tipo de dato de número entero a número entero positivo. Además, se modificó el código fuente del servidor para que el error devuelto en este

caso sea un error controlado.

- Un problema realizando pruebas de configuración del servidor de caché, el cual no estaba disponible durante la ejecución de las mismas. Este problema no había sido encontrado antes porque durante las pruebas manuales siempre se usaban direcciones IP accesibles, pero las pruebas con generadores generados automáticamente también producen direcciones IP no existentes, lo cual provocaba un error no controlado en el servidor cuando estas direcciones IP no existentes se usaban como servidor de caché. El error fue solucionado en el servidor.

Como se observa, todos los defectos encontrados están relacionados con el uso de los generadores de datos, lo cual revela su importancia en las pruebas basadas en propiedades. Sin embargo, aunque pueda parecer que los generadores de datos totalmente aleatorios son buenos porque ayudaron a detectar defectos no encontrados con otros escritos manualmente, si se atiende a otros parámetros, como es la cobertura de las pruebas en el código fuente del servidor, la conclusión a la que se llega es bastante diferente. Así, con los generadores escritos manualmente en la aproximación en la que la que la implementación de la máquina de estados QuickCheck también se escribía manualmente, se obtenía una cobertura de líneas de código del 94 %. En cambio, usando los generadores de datos generados por la herramienta, la cobertura se reduce al 69 %.

La reducción de la cobertura de código se explica porque, mientras que con los generadores de datos escritos manualmente se generan combinaciones específicas de los valores de entrada, ahora los valores que se generan son puramente aleatorios, teniendo en cuenta únicamente los tipos de datos definidos en la especificación WSDL para los parámetros de entrada de cada operación. De la misma forma que se ha comentado en la sección 6.5.6 del capítulo 6 con referencia a las APIs de integración, esta aproximación podría no ser del todo útil en las pruebas en muchos casos. Por un lado, puesto que siempre se generan valores teniendo en cuenta los tipos de datos definidos en la especificación WSDL, esta técnica no permite generar valores de un tipo de dato no esperado por la operación a probar para comprobar qué sucedería en estos casos, por ejemplo, invocar una operación con un `string` cuando se espera un `integer` y comprobar que se devuelve el error esperado. Esto implica que, con los generadores de datos por defecto, este tipo de casos de prueba negativos no van a ser utilizados con esta aproximación. Por otro lado, generar datos completamente aleatorios, sin tener en cuenta otros valores generados anteriormente, puede no ser lo más conveniente en algunos casos. En cualquier caso, el hecho de encontrar defectos ocultos y obtener una cobertura de código más baja ilustra que quizá la mejor aproximación para obtener unos buenos generadores de datos es partir de los generadores generados automáticamente e ir refinándolos para mejorar la cobertura del código paulatinamente.

Por otro lado, esta técnica, si bien es cierto que requiere escribir menos código

fuente para realizar las pruebas que el resto de aproximaciones comentadas en este capítulo, y ayudó a encontrar defectos ocultos hasta ahora, no se valoró lo suficientemente bien como parte del piloto realizado en la empresa Interoud Innovation. La razón principal se debe al uso del lenguaje OCL, el cual no parece ser lo suficientemente amigable en la programación de las pruebas del software. Además, si los generadores producidos por defecto por las herramientas WSToolkit no fueran reemplazados por los generadores específicos, estos defectos habrían aparecido igualmente con la aproximación en la que la máquina de estados se implementa manualmente (a partir de un esqueleto producido automáticamente).

No obstante, se debe tener en cuenta que en Interoud Innovation se está usando Erlang y QuickCheck, y, por tanto, los programadores están familiarizados con las pruebas basadas en propiedades desde hace años, lo cual puede provocar que estos programadores prefieran esta forma de programar las pruebas en vez del uso de un nuevo lenguaje como es OCL. Podría decirse que el esfuerzo de escribir la semántica de las operaciones en el lenguaje OCL puede considerarse similar a escribir la semántica como precondiciones y postcondiciones QuickCheck para un programador que no es un experto ni en las pruebas basadas en propiedades, ni en el lenguaje OCL.

7.5. Resumen

Este capítulo presenta una técnica para probar servicios web usando una aproximación basada en propiedades. Esta técnica se basa en la ya descrita en el capítulo 6 para probar APIs de integración, pero se aprovecha del hecho de que ahora el sistema a probar es un servicio web para automatizar algunas partes más del proceso de pruebas. Así, el adaptador que conecta los casos de prueba con el sistema a probar puede ser generado automáticamente a partir de una especificación del servicio web, al contrario de lo que ocurría para las APIs de integración genéricas. Para esto, se usa una especificación WSDL que describe las operaciones que forman parte de dicho servicio web, así como los tipos de datos utilizados en las entradas y respuestas de las operaciones a través de un esquema XSD.

Con respecto a la especificación de pruebas, aunque existen servicios web sin estado en los que es posible usar propiedades independientes para probar cada una de las operaciones de forma independiente, lo más común es que la invocación de las operaciones produzca efectos colaterales que afecten al resultado de otras operaciones, es decir, lo más normal es que el componente que implementa un servicio web tenga estado. Por ello, de la misma forma que se explicó para probar APIs de integración, lo más adecuado para construir la especificación de pruebas basada en propiedades, en este caso, es usar máquinas de estados QuickCheck.

En este capítulo se han presentado dos aproximaciones para construir esta especificación de pruebas. Por un lado, es posible generar un esqueleto de la máquina

de estados a partir de la especificación WSDL. Por otro lado, las herramientas construidas soportan el uso del lenguaje OCL para añadir información semántica a la especificación WSDL y, generar, con toda esta información, la máquina de estados QuickCheck completa. Además, en el primero de los casos, también se ofrecen herramientas que ayudan a cambiar el código de pruebas cuando se modifica la sintaxis del servicio web. Obviamente, en el segundo caso, esto no es necesario, puesto que la especificación de pruebas se generaría de nuevo sin coste. Por otro lado, y en todo caso, si cambia la implementación del servicio web, pero se respeta la sintaxis y semántica del mismo desde un punto de vista externo, la especificación de pruebas no cambiará, puesto que ésta es una especificación de pruebas abstracta que puede probar cualquier implementación del servicio web.

Ambas aproximaciones han sido evaluadas con un caso de estudio, en concreto, el servicio web de gestión de información relacionada con el subsistema OSS/BSS que proporciona el componente VoDKATV-core del sistema VoDKATV. Una primera conclusión que se extrajo con este caso de estudio es la gran cantidad de código fuente que ya no se necesita escribir con cualquiera de las dos aproximaciones. Así, en la primera de ellas, el código del adaptador se genera automáticamente a partir de la especificación WSDL, lo cual, además de ahorrar esfuerzo, elimina los errores humanos que pueden ocurrir si esta tarea se realiza manualmente. Por su parte, en la segunda aproximación, además del adaptador, también se genera la máquina de estados QuickCheck a partir de una especificación WSDL con restricciones OCL. Se observó con el caso de estudio que el código escrito OCL es mucho menos que el de máquina de estados QuickCheck generada. Sin embargo, aunque con la segunda aproximación es posible ahorrar muchas líneas de código escritas manualmente, existe un cierto rechazo al uso del lenguaje OCL en la industria, al menos en el caso en el que ya se conoce el lenguaje de programación Erlang y la herramienta QuickCheck. A modo resumen, la tabla 7.1 muestra la cantidad de código que se tenido que escribir manualmente y cuánto código ha sido generado automáticamente con estas dos aproximaciones con el caso de estudio considerado, comparando además estos valores con las líneas de código de la máquina de estados QuickCheck existente antes de usar estas dos aproximaciones.

Por otro lado, a partir de este caso se estudio se pudo observar que los generadores de datos son muy importantes cuando se usa una aproximación basada en propiedades. Así, mientras que las comprobaciones que se realizaron en dos aproximaciones distintas eran las mismas, con una se encontraron errores y con la otra no. Esto fue debido a que se utilizaron generadores de datos diferentes para cada aproximación. Se puede deducir así que una posible aproximación para refinar estos generadores de datos es comprobar la cobertura del código a medida que se ejecutan las pruebas.

Se debe destacar que con ambas aproximaciones se consigue que los probadores de software únicamente centren su trabajo en definir las comprobaciones que se deben realizar para comprobar que el servicio web funciona de la manera esperada,

Aproximación	Código de pruebas escrito	Código de pruebas generado
PBT inicial	Máquina de estados QuickCheck para 20 operaciones (1,1K LOC) y adaptador para dichas operaciones (400 LOC)	-
WSToolkit	Implementación de las precondiciones, postcondiciones y funciones de cambio de estado para 20 operaciones en el esqueleto de la máquina de estados QuickCheck (950 LOC)	Esqueleto de máquina de estados QuickCheck para 363 operaciones, es decir, todas las operaciones del WSDL (2,5K LOC), y adaptador para dichas operaciones (3,4K LOC)
WSDL y OCL	Especificación OCL para 20 operaciones (519 LOC)	Adaptador para 363 operaciones, es decir, todas las operaciones del WSDL (3,4K LOC), y máquina de estados QuickCheck para 363 operaciones, 20 de ellas con precondiciones, postcondiciones y funciones de cambio de estado (3,7K LOC).

TABLA 7.1: *Comparación de LOC entre diferentes aproximaciones usadas para probar servicios web*

es decir, en la definición de las precondiciones y postcondiciones asociadas a cada operación del servicio web, ya sea en el lenguaje Erlang o en OCL. Esto se consigue gracias a la automatización de todas las partes posibles del proceso de pruebas, lo que permite mejorar la eficiencia y la eficacia en la realización de las mismas.

Las herramientas desarrolladas para poder llevar a cabo las técnicas explicadas en este capítulo han sido las siguientes:

- Marco de trabajo WSToolkit (en colaboración con Huiqing Li de la Universidad de Kent): `https://github.com/RefactoringTools/WSToolkit`.
- Herramienta de generación de propiedades y máquinas de estados QuickCheck a partir de una especificación WSDL y OCL (en colaboración con Macías López y Henrique Ferreño del grupo MADS de la Universidad de A Coruña): `https://github.com/hferreiro/WSDL-OCLToErlang`.

Finalmente, comentar que con esta aproximación es posible probar servicios web escritos en cualquier lenguaje y, aunque se ha usado una especificación WSDL con servicios web que devuelve datos en el lenguaje XML, sería posible extender esta aproximación para poder ser usada con otro tipo de lenguajes de intercambio, como es el caso de JSON. De hecho, relacionado con el uso de JSON, como parte del proyecto PROWESS, se ha construido una herramienta, jsongen [113, 191], que funciona de manera similar a la técnica descrita en este capítulo. La diferencia radica en que jsongen usa JSON *Schemas* [39] extendidos, que especifican enlaces entre las operaciones del servicio web usando JSON *Hyper-Schema*, para describir el servicio web a probar. Estos esquemas se complementan con una implementación reducida de una máquina de estados QuickCheck en Erlang donde se especifican las precondiciones y postcondiciones para cada operación. Los esquemas, junto con la máquina de estados QuickCheck, substituyen a la especificación WSDL con restricciones OCL presentada en este capítulo. Relacionados con esta aproximación, se han realizado experimentos de uso de esta herramienta con VoDKATV, en concreto, con el servicio web que usan los usuarios (*set-top-boxes*, ordenadores, tabletas, etc.) para acceder al sistema, el cual maneja datos formateados en el lenguaje JSON [88].

8

INTEGRACIÓN DE COMPONENTES

8.1. Introducción

Los sistemas software complejos suelen estar formados por varios componentes que se integran entre sí. El proceso de integración define cómo se deben conectar los diferentes componentes que forman parte de un sistema informático para que éstos funcionen, de forma conjunta, de la manera deseada. Cada uno de estos componentes representa una unidad de software indivisible dentro del sistema. Además, en este trabajo se considerará que dicha unidad de software ha sido probada usando las técnicas descritas en los capítulos 5 y 6 (o el capítulo 7 si éste ofrece un servicio web).

Los componentes software interactúan entre sí formando un sistema informático que cumple una serie de requisitos establecidos. Este proceso de integración puede ser más o menos sencillo en función de cómo estén diseñados los componentes que se integran. Así, existen dos tipos de medidas que se deben tener en cuenta en este tipo de sistemas [282, 286]: la complejidad de cada módulo y la complejidad en la interconexión entre módulos.

Teniendo en cuenta estas dos medidas de complejidad, la división en componentes del sistema software debería cumplir las siguientes condiciones: minimizar el *acoplamiento* entre componentes y maximizar la *cohesión* dentro de cada componente [373]. Minimizar el acoplamiento entre componentes provocará un menor número de dependencias e interconexiones entre ellos. Con respecto a la cohesión, si ésta se maximiza, se conseguirá una mejor definición de los diferentes componentes, facilitando el mantenimiento del sistema. Cabe mencionar que existen enfoques

que utilizan estas medidas como base para realizar las pruebas de integración [220].

Una de las aproximaciones que minimizan el acoplamiento entre componentes es el uso de fachadas [193] de acceso que definan APIs de integración. De esta forma, cada componente o unidad de software indivisible se comunicará con otros componentes a través de las APIs de integración que éstos últimos ofrecen (ver figura 8.1). A su vez, estas APIs de integración pueden ser probadas con las técnicas explicadas en el capítulo 6 (o el capítulo 7 si dicha API es un servicio web).

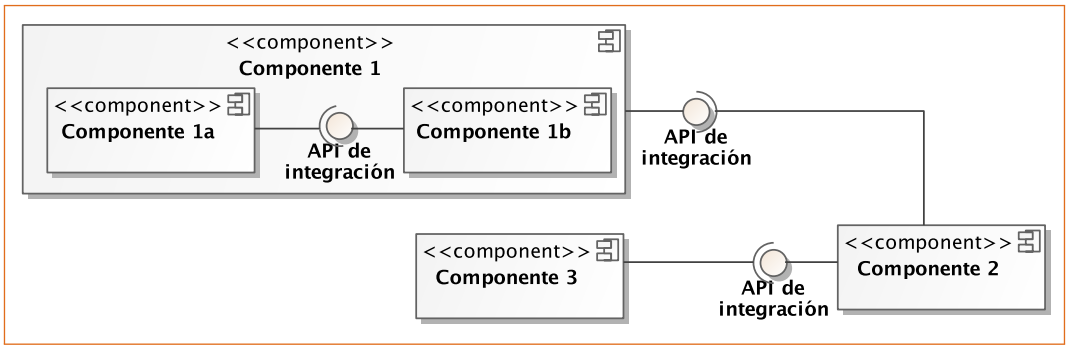


FIGURA 8.1: *Integración de componentes software a través de APIs de integración*

De todas formas, cuando varios componentes se integran, también debe ser probada la comunicación entre ellos, es decir, que el uso de la API de integración que ofrece un componente es el adecuado por parte de otro componente dado. Este capítulo describe una aproximación de pruebas que permite realizar exactamente esta actividad, es decir, comprobar que la integración entre componentes funciona de la manera esperada cuando esta integración se realiza a través del uso de una API de integración definida y probada. Para ello, se usará una aproximación basada en propiedades, las cuales describirán cómo debe ser la comunicación entre dos componentes.

El resto del capítulo se estructura de la siguiente manera. La sección 8.2 explica los conceptos básicos de las pruebas de integración. Posteriormente, en la sección 8.3, se describe la aproximación propuesta para probar integraciones entre componentes usando propiedades, ilustrando cómo se ha usado esta aproximación con un caso de estudio real. Por último, en la sección 8.4, se resumirán los contenidos de este capítulo.

8.2. Las pruebas de integración

Cuando un sistema software está estructurado en diferentes componentes, cada una de estas unidades de software deben ser probadas de forma independiente para asegurar que realizan, de la manera esperada, las tareas para las que están diseñadas. Sin embargo, estos componentes también deben ser probados de forma conjunta

para comprobar que interactúan entre ellos de la forma deseada. Este último tipo de pruebas es lo que se conoce como pruebas de integración.

Mientras que las pruebas de unidad tratan únicamente de comprobar el comportamiento de una unidad de software independiente, las pruebas de integración combinan varias unidades de software que funcionan juntas para evaluar los resultados de dicha integración, y comprobar si éstos son los esperados. De esta forma, este tipo de pruebas se encargan de comprobar la comunicación entre dos componentes a través de las interfaces que éstos proporcionan. Así, las pruebas de integración constituyen el nivel de pruebas del software en el que los componentes software individuales son combinados y probados como un grupo.

Las pruebas de integración definen una progresión ordenada de pruebas en las que los componentes software se combinan hasta que todo el sistema se haya integrado por completo. Así, a la hora de realizar la integración de componentes para implementar un sistema software, existen varias estrategias que pueden ser usadas. Una posible opción es realizar una integración de todos los componentes en un único paso, también conocido como integración *big-bang* [110]. Por otro lado, también es posible usar una aproximación *bottom-up*, en la que se comienza integrando aquellos componentes que ofrecen una funcionalidad más básica para, posteriormente, ir integrando componentes de más alto nivel; o una aproximación *top-down*, en la que se realiza el proceso contrario, es decir, la integración comienza con los componentes de alto nivel y, poco a poco, se se va añadiendo funcionalidad de bajo nivel hasta poder completar el sistema completo. Existen otras estrategias como comenzar primero integrando los componentes más críticos, o realizar las integraciones según las funcionalidades que se vayan necesitando [207]. En cualquier caso, de la misma forma que existen diferentes estrategias para realizar la integración entre los diferentes componentes de un sistema software, también existen estrategias similares para realizar las pruebas de integración de los mismos [172]. La estrategia elegida para realizar la integración entre componentes afecta a cómo se deben realizar las pruebas de integración, y al coste de las mismas [307].

Así, por ejemplo, en la aproximación *big-bang*, tanto la integración como las pruebas de integración comienzan después de finalizar la implementación de todos los componentes, con lo cual cual se convierte en una fase muy tardía en el desarrollo, además de que puede ser difícil identificar la fuente de los fallos que aparezcan en dichas pruebas, las cuales se pueden complicar en exceso. De la misma forma, usar una estrategia *bottom-up*, aunque tiene la ventaja de que las integraciones entre los componentes de bajo nivel puede ser probada en fases tempranas del proyecto, la funcionalidad completa es probada muy tarde, por lo que si aparecen problemas de diseño o de eficiencia sólo pueden ser detectados en la recta final de la fase de integración. Por el contrario, usar una estrategia *top-down* no tiene este problema, pero probar las integraciones entre componentes requiere componentes no reales (componentes de reemplazo) que simulan el comportamiento de componentes de

bajo nivel, todavía no integrados en el sistema.

Además, existen muchas situaciones en las que estas técnicas no se ajustan al proyecto, por ejemplo, si no existe un sistema jerárquico de componentes o si las fechas de finalización de los diferentes componentes no es predecible. Por supuesto, en estos casos, siempre es posible realizar una integración *ad-hoc*, en la que los componentes son integrados a medida que son finalizados o cuando sea razonable llevar a cabo dicha integración. En este caso, el uso de componentes de reemplazo también suele ser necesario.

Por tanto, comprobar la integración entre dos componentes usando únicamente uno de ellos y prescindiendo del otro suele ser una aproximación útil en muchos casos por varios motivos. En este caso, como se muestra en la figura 8.2, el componente que se prueba (SUT), usa otro componente, conocido con el nombre de *componente dependiente* (DOC), que se reemplaza durante la integración.

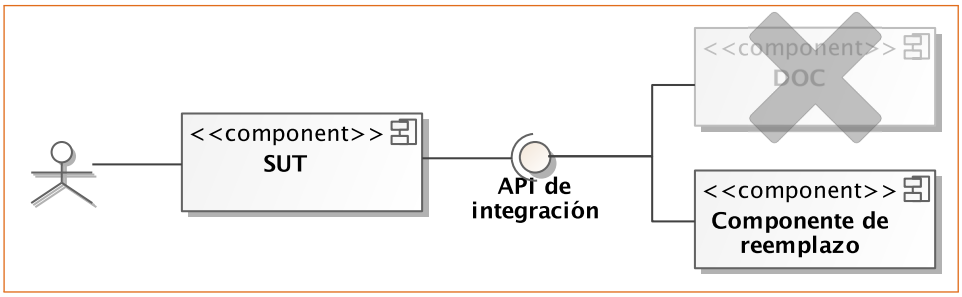


FIGURA 8.2: *Integración de componentes: SUT y DOC*

Puesto que en el desarrollo de software de sistemas complejos, el diseño e implementación de componentes software es, a menudo, dividido en varias fases, muchas de ellas paralelas, y siendo el tamaño, complejidad y prioridad de cada componente normalmente diferente, podría ocurrir que mientras que un componente puede estar en fase de implementación, otro componente podría estar en fase de integración con otros componentes software. Por tanto, en muchos casos, no es posible probar la integración de dos componentes combinando los dos componentes reales, simplemente porque uno de ellos todavía no está disponible. Por otro lado, usar los dos componentes reales que se integran en las pruebas de integración, al menos en una fase inicial, puede complicar el análisis de los problemas encontrados durante las pruebas, puesto que éstos podrían ocurrir por fallos en la propia integración o bien por defectos en el componente dependiente que no tienen nada que ver con la integración en sí misma.

De esta forma, substituir el componente dependiente real por otro componente especialmente diseñado para las pruebas para comprobar que una integración está siendo implementada de la forma apropiada, además de provocar respuestas más rápidas, evita efectos colaterales y, evidentemente, permite la posibilidad de ejecutar

pruebas de integración incluso antes de que el propio componente dependiente esté finalizado (o incluso comenzado) [212, 213, 284].

Existen varios tipos de aproximaciones que se pueden utilizar para realizar este reemplazo [264]:

- **Componentes *dummies***: este tipo de componentes se usan cuando únicamente es necesario proporcionar definición de interfaces, como, por ejemplo, cabeceras de funciones que pueden ser necesarias para compilar un módulo.
- ***Stubs***: los *stubs* son componentes que reemplazan un componente original proporcionando respuestas predefinidas para las llamadas a las operaciones requeridas en las pruebas.
- ***Espías***: los componentes *espías* se utilizan para reemplazar un componente, proporcionando respuestas para las llamadas a las operaciones requeridas de la misma forma que los *stubs*, pero registran internamente las llamadas realizadas para, posteriormente, poder proporcionar información acerca de las mismas y, de esta forma, permitir a otros componentes de pruebas comprobar si se han realizado las llamadas esperadas.
- ***Mocks***: estos componentes reemplazan el sistema original, proporcionan respuestas para las llamadas a operaciones comprobando ellos mismos que las llamadas que el sistema está realizando se corresponden exactamente con las llamadas esperadas en el componente real, pudiendo tener en cuenta incluso el orden en el que las operaciones son invocadas.
- **Componentes *ficticios***: un componente *ficticio* es una implementación alternativa, mucho más ligera y sencilla, de un componente real, pero que simula el comportamiento de dicho componente real, sin llevar a cabo las acciones reales que éste realiza. Algunas situaciones en las que este tipo de componentes suelen ser usados son, por ejemplo, cuando el componente reemplazado no ha sido construido todavía, es demasiado lento, o no está disponible en el entorno de pruebas.

Con el uso de este tipo de componentes (cuyas características se resumen en la tabla 8.1), en concreto, con el uso de *stubs*, espías, *mocks* o componentes ficticios, es posible implementar las pruebas de integración componentes incluso antes de que los dos componentes que se integran estén finalizados. En este caso, únicamente el componente a probar, nombrado como SUT, es necesario, puesto que el componente dependiente (DOC) es reemplazado por otro componente especialmente creado para las pruebas. Evidentemente, una vez terminadas las pruebas siguiendo esta aproximación, y finalizada la implementación tanto del componente a probar como del componente dependiente, se deben repetir las mismas pruebas de integración para comprobar que, usando los componentes reales, las comprobaciones realizadas siguen siendo ciertas.

Aproximación	Facilita compilación	Proporciona respuestas	Registra llamadas	Comprueba llamadas
Componentes *dummies*	✓			
Stubs	✓	✓		
Espías	✓	✓	✓	
Mocks	✓	✓	✓	✓
Componentes *ficticios*	✓	✓		

TABLA 8.1: *Características de los diferentes tipos de componentes de reemplazo*

8.3. Uso de propiedades para probar la integración de componentes

Esta sección presenta una metodología de pruebas para comprobar la integración entre dos componentes [137–139]. Esta aproximación está basada en el uso de propiedades y, en concreto, se usará la herramienta QuickCheck para mostrar cómo funciona.

Aunque es posible que, en algunos casos, el uso de propiedades independientes sea suficiente para realizar este tipo de pruebas, en muchas ocasiones, el uso de máquinas de estados es más adecuado. Esto es debido a que la mayoría de las veces, la forma de realizar estas pruebas es invocando operaciones relacionadas de un componente, es decir, con efectos colaterales entre ellas, y comprobando, por un lado, si la ejecución de estas operaciones produce las invocaciones de operaciones esperadas en otro componente dependiente, y, por otro lado, si los valores que este componente dependiente devuelve son procesados de la manera esperada por el componente inicial.

Por esta razón, se explicará la técnica propuesta usando máquinas de estados, en concreto, máquinas de estados QuickCheck, pero será totalmente aplicable al uso de propiedades independientes aplicables en escenarios más simples con ausencia de efectos colaterales.

8.3.1. Arquitectura de pruebas

Cuando las pruebas que se quieren realizar son pruebas de integración, el principal objetivo es comprobar que dos (o más) componentes que interactúan entre sí lo hacen de la forma esperada. Así, lo que se quiere comprobar es que cuando se necesita algún servicio que un componente proporciona (componente SUT), y este servicio, su vez, necesita interactuar con otro componente (DOC) para llevar a

cabo alguna operación, el primero de los componentes invoca las operaciones adecuadas en el otro componente y, además, éste procesa los datos devueltos por el componente dependiente de la manera esperada.

Por tanto, para realizar este tipo de comprobaciones, técnicamente no es necesario disponer del segundo componente, puesto que no se está probando la funcionalidad, sino únicamente la interacción entre ellos. Lo que realmente se necesita es un componente con el que el propio componente a probar pueda interaccionar de la misma forma que lo haría con el componente dependiente real. Este componente puede reemplazar al componente dependiente original, pero debe ofrecer la misma interfaz (o un subconjunto) que éste, en concreto, la parte de la interfaz usada por el componente a probar, proporcionando los mismos tipos de respuestas para las operaciones. Obviamente, este componente de reemplazo no debe replicar la funcionalidad del componente original, porque sería duplicar esfuerzo de implementación, sino que únicamente debe simular su comportamiento proporcionando respuestas, muchas veces predefinidas, para las operaciones que proporciona.

De esta forma, no importa si el componente de reemplazo ofrece realmente el servicio que proporciona, o si las respuestas que devuelve son simplemente simuladas. Lo realmente importante es que este componente de reemplazo ofrezca las mismas operaciones que el componente a probar usa del componente reemplazado, las cuales deben recibir los mismos parámetros, y devolver los mismos valores que el componente dependiente original. Por tanto, para los componentes software, este componente de reemplazo debe ser equivalente al componente original, pareciendo que funciona de la misma forma que éste, aunque en realidad no realice ninguna acción real. Construir este tipo de componentes debe ser una tarea fácil y, además de evitar los problemas comentados en la sección 8.2 acerca de la disponibilidad de componentes para comenzar con las pruebas de integración, permite obtener respuestas más rápidas, acelerando, por tanto, la ejecución de las pruebas, y evitando efectos colaterales indeseados en la ejecución de las mismas.

En cualquier caso, para comprobar realmente la integración entre el primer componente (SUT) y el segundo (DOC), que es el objetivo principal de las pruebas de integración, es necesario algo más. La propuesta de esta aproximación es que el componente de reemplazo registre todos los accesos a las operaciones que proporciona, en concreto, se deberá registrar el nombre de la operación invocada, los argumentos que recibió y el resultado que devolvió. Además, este componente también debe ofrecer una forma de obtener esta información registrada. Así, de entre todos los tipos de componentes de reemplazo explicados en la sección 8.2, en este caso se usará un componente *espía*. De esta forma, la traza de llamadas puede ser recuperada una vez realizadas las llamadas oportunas en el componente a probar, y así, comprobar que se han producido las interacciones esperadas entre ambos componentes. Por tanto, para cada operación a probar hay una forma de comprobar qué operaciones han sido llamadas en el componente dependiente.

Teniendo en cuenta estas consideraciones, la figura 8.3 muestra un diagrama con los componentes que forman parte de esta arquitectura de pruebas: el componente a probar (SUT), el componente *espía* que reemplaza al componente dependiente (DOC) y registra las trazas, y la especificación de pruebas QuickCheck, la cual invoca las operaciones del componente a probar y examina las traza de operaciones invocadas en el componente de reemplazo. Este tipo de pruebas no se corresponden realmente con pruebas de caja negra o pruebas de caja blanca, sino que pueden considerarse como pruebas de caja gris [296], puesto que, por un lado, se está probando el sistema desde una perspectiva externa, pero, por otro lado, es necesario conocer las interacciones que dicho componente a probar realiza con los componentes dependientes.

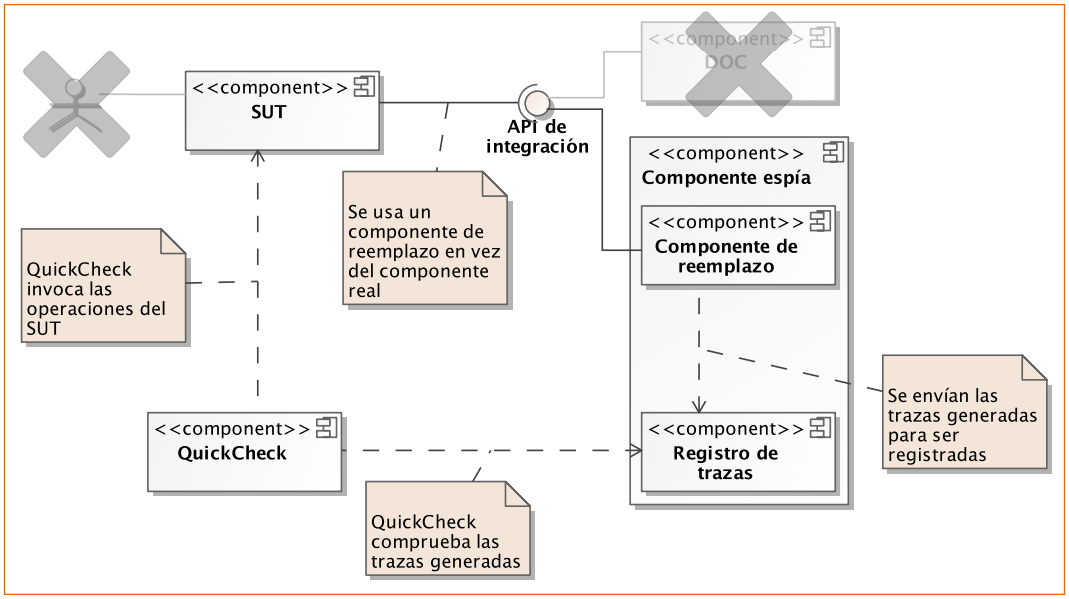


FIGURA 8.3: *Arquitectura propuesta para realizar pruebas de integración*

Si se usa una máquina de estados QuickCheck en la que se incluyan las diferentes operaciones que ofrece el componente a probar como comandos posibles de dicha especificación de pruebas, cuando ésta se ejecuta, se generarán secuencias de operaciones posibles, que serán los casos de prueba usados. Cuando estas operaciones son ejecutadas, se invocarán las operaciones del componente a probar incluidas como comandos en dicha máquina de estados, y cuando esto ocurre, se producirán una serie de interacciones entre dicho componente (el componente SUT) y otro componente dependiente (el componente DOC). Estas interacciones son registradas en el componente *espía* usado para reemplazar a dicho componente dependiente. De esta forma, estas interacciones pueden ser inspeccionadas, y así, comprobar que se han producido las llamadas esperadas en todos los escenarios posibles.

Estas comprobaciones formarán parte de las postcondiciones de la máquina de

estados QuickCheck. De esta forma, además de comprobar otros aspectos como el resultado devuelto por el componente a probar para una operación concreta en un estado concreto, se podrá comprobar si dicho componente invocó las operaciones adecuadas con los argumentos adecuados en el componente dependiente. Puesto que la traza almacenada contiene el nombre de las operaciones, sus argumentos y el valor devuelto, es posible realizar este tipo de comprobaciones.

8.3.1.1. Pruebas de integración negativas

El procedimiento de pruebas descrito hasta ahora se basa en la implementación de componentes *espía* que ofrecen la misma interfaz y simulan el mismo comportamiento que los componentes originales y, de esta forma, comprobar que la comunicación entre dos componentes ha sido implementada de forma adecuada. Sin embargo, existen un gran número de escenarios de fallo que caen fuera del alcance de la aproximación descrita hasta ahora. Cuando se prueba la integración entre componentes, hay cuestiones relevantes que se deben tener en cuenta como, por ejemplo, qué sucedería si el componente dependiente DOC falla al responder una petición realizada por el componente a probar SUT, o si existe un problema en la red que impide el envío y recepción de los mensajes entre los componentes.

Hasta ahora, únicamente se han tenido en cuenta las pruebas positivas, pero, de esta forma, no es posible considerar este tipo de situaciones de fallo, en las que la comunicación podría fallar debido a cualquier tipo de circunstancia, bien interna del componente o bien externa al componente, como puede ser un problema de red o similar. Teniendo en cuenta este tipo de situaciones es posible comprobar si el componente se comporta de la manera esperada ante fallos en la comunicación con el componente dependiente.

Para tratar con pruebas de integración negativas, una opción es crear nuevos comandos en la máquina de estados QuickCheck que fuercen situaciones de fallo, por ejemplo, retrasar las respuestas del componente de reemplazo, o incluso forzar a que el componente de reemplazo no ofrezca ninguna respuesta al componente llamador. De esta forma, es posible simular situaciones anómalas en la comunicación, es decir, situaciones no previstas en un escenario positivo. Esta técnica se conoce como patrón *saboteador* [264], puesto que su propósito es inyectar respuestas inválidas y errores en la comunicación de un componente externo con el componente a probar.

Realizar este tipo de pruebas requiere la modificación de la especificación de pruebas para incluir los comandos que fuercen errores, pero también es necesario añadir soporte al componente de reemplazo para generar este tipo de errores en situaciones controladas. Además, cuando se generan y ejecutan secuencias de prueba con estos nuevos comandos de error, las postcondiciones podrían ser diferentes a las existentes para las pruebas positivas, puesto que la traza esperada, en este caso, puede no ser la misma que la esperada cuando no hay errores en la comunicación.

Por tanto, se debe definir cómo debe reaccionar el componente a probar en estos casos, y comprobar que el comportamiento es el esperado en las postcondiciones de cada operación.

8.3.1.2. Integración de los dos componentes reales

La ejecución de los casos de prueba proporciona una cierta seguridad de que para cada operación del componente a probar se realizan las llamadas esperadas al componente dependiente correspondiente. En cualquier caso, aunque este tipo de pruebas está probando la integración entre componentes, y éstos, a su vez, también deben ser probados de forma independiente a través de las pruebas de unidad (ver capítulo 5) y de componente (ver capítulo 6 y 7), estas pruebas se están realizando usando un componente de reemplazo, y no el componente real, lo cual implica que la ejecución de estas pruebas no garantiza que el sistema completo funcione de la manera esperada.

Sin embargo, esta técnica soporta una variación que transforma el componente *espía* en un *proxy-espía*, la cual se representa en la figura 8.4. Así, el componente de reemplazo, en vez de devolver datos, quizá ficticios, para las operaciones invocadas, realizará la misma llamada en el componente dependiente original DOC, y devolverá el resultado que éste proporcione. El motivo por el que resulta recomendable usar este componente intermedio es que todavía puede ser necesario analizar las trazas de operaciones invocadas por el componente a probar. De esta forma, la especificación de pruebas será exactamente la misma que en el caso anterior. Un error en estas pruebas, después de haber ejecutado las pruebas con el componente *espía*, desvelará un error en el canal de comunicación entre los dos componentes reales.

La figura 8.5 muestra un diagrama de secuencia en el que se aprecian las diferencias entre esta aproximación y usar simplemente un componente *espía*. Así, cuando se usa un componente *espía* (pasos 1–7 del diagrama de secuencia de la figura 8.4), éste genera los datos que se devuelven al componente a probar. Sin embargo, al usar un componente *proxy-espía* (pasos 8–16 del mismo diagrama de secuencia), los datos que devuelve el componente de reemplazo los obtiene directamente del componente dependiente real. En cualquier caso, en ambas situaciones, cuando se quiere comprobar que la traza de operaciones invocadas es la esperada, QuickCheck tiene que realizar las mismas acciones, es decir, consultar dichas trazas y compararlas con las trazas esperadas (pasos 17–19 del diagrama de secuencia).

Finalmente, una vez superados los dos tipos de pruebas explicados anteriormente, es posible eliminar el componente *espía* y el componente *proxy-espía*, y ejecutar la especificación de pruebas usando los dos componentes reales integrados de la misma forma que en el sistema real. En este caso, la especificación de pruebas no puede comprobar la traza de operaciones llamadas por el componente a probar,

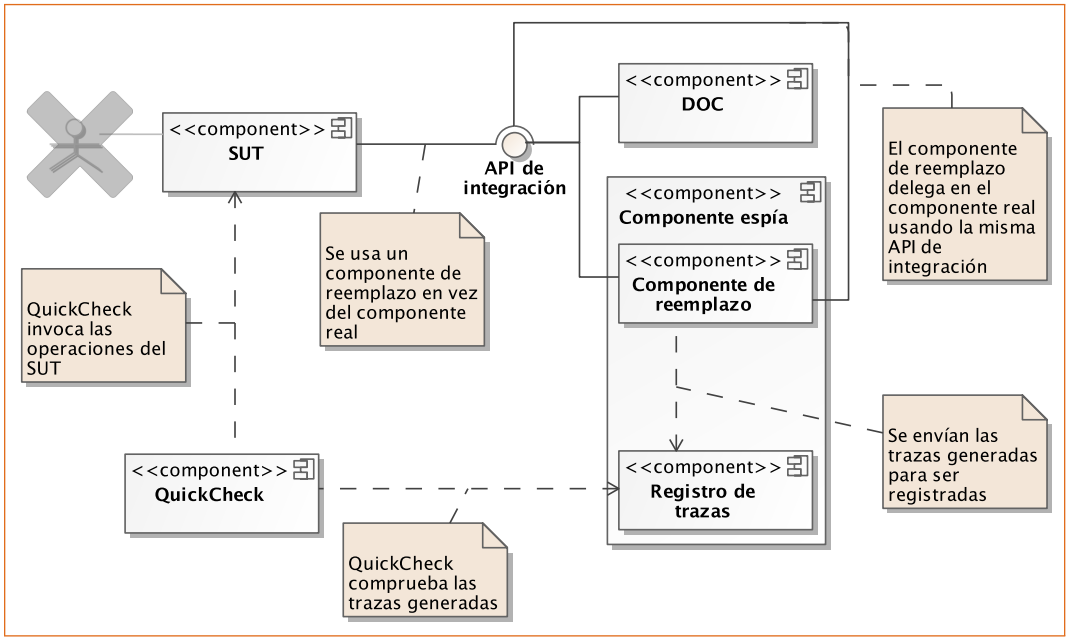


FIGURA 8.4: *Arquitectura propuesta para realizar pruebas de integración usando los dos componentes reales*

pero sí es posible comprobar que el componente a probar, usando el componente dependiente original, devuelve los resultados esperados.

8.3.2. Caso de estudio: programador de directos y grabaciones LiveScheduler

LiveScheduler es un componente software, implementado en Erlang, que se integra dentro de la arquitectura de VoDKATV (ver figura 3.6 de la página 56), y se usa para programar eventos multimedia que serán emitidos a través de Internet. Los administradores de la plataforma LiveScheduler pueden definir los contenidos multimedia que se programan para ser emitidos a través de un canal (*multicast* o *unicast*). Así, como se muestra en la figura 8.6, la información manejada por el componente LiveScheduler se organiza en dos tipos de entidades:

- **Eventos,** que son los contenidos multimedia que se emitirán a través de Internet y, opcionalmente, pueden ser grabados a un fichero local para poder ser conservados. Normalmente, los eventos son grabaciones que se están realizando a la vez que se emiten por Internet, por ejemplo, una rueda de prensa, un partido de fútbol, una conferencia, etc. También es posible definir eventos periódicos que se emitirán cada cierto tiempo, por ejemplo, todos los martes de 8 de la mañana a 10 de la mañana. Cada evento puede tener asociados medios de publicidad que pueden ser reproducidos antes, después o durante el evento como cortes de publicidad.

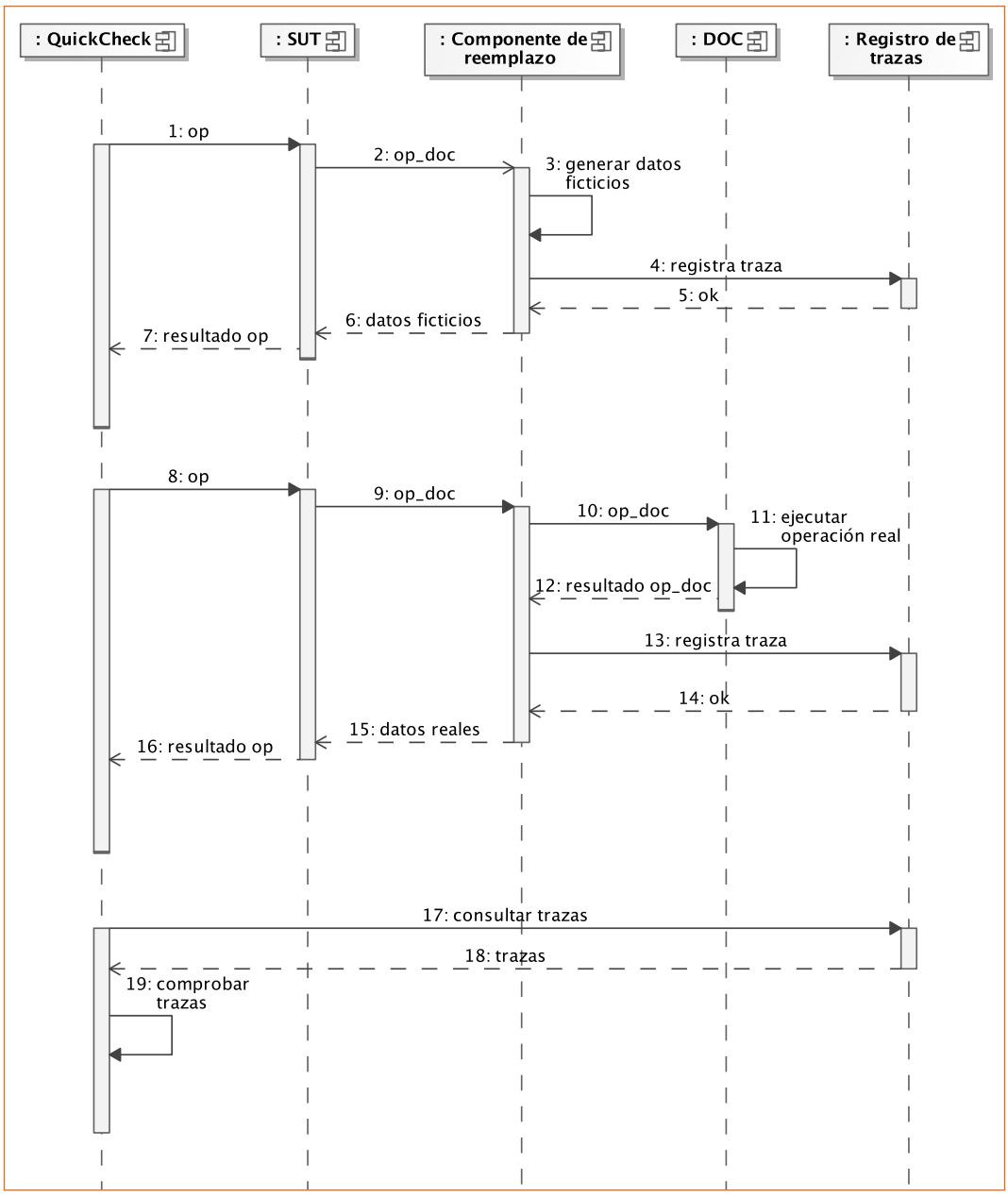


FIGURA 8.5: *Ejemplo de interacciones entre componentes en las pruebas de integración*

- **Emisiones,** que representan un grupo de eventos que no se solapan y que son emitidos a través del mismo canal multimedia de salida. Si no hay un evento actual para una emisión, el componente LiveScheduler programará un contenido de relleno para dicho canal.

Los datos de los eventos y emisiones introducidos por el administrador se almacenan en una base de datos local manejada por el propio componente LiveSche-

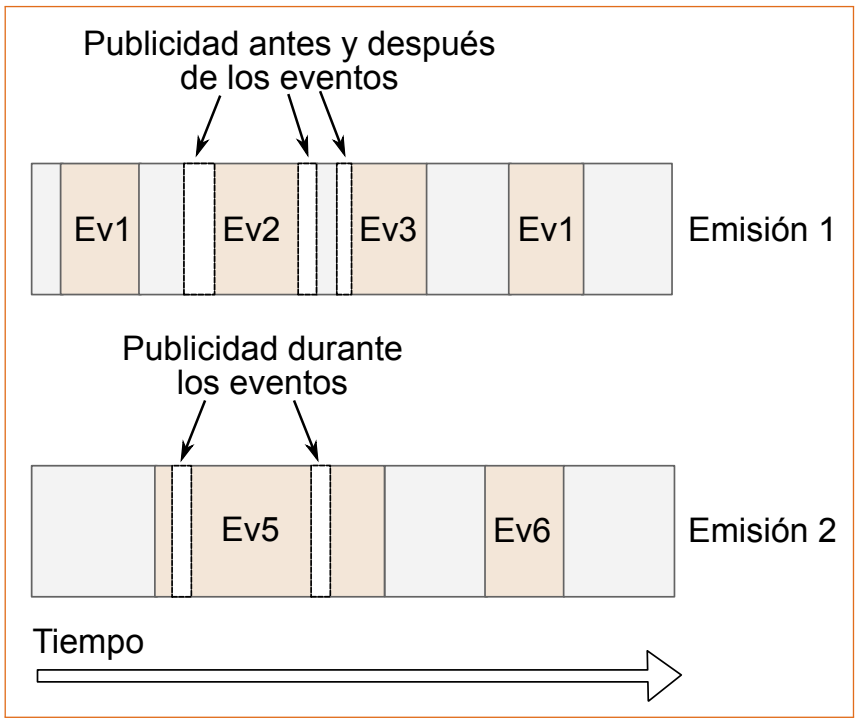


FIGURA 8.6: *El programador LiveScheduler: eventos y emisiones*

duler. De esta forma, el programador LiveScheduler, además de almacenar estos datos, los procesa para poder comenzar o detener las emisiones de los eventos correspondientes en el momento oportuno (invocando internamente las operaciones `force_start_event`, `force_stop_event`, `force_start_advertising`, etc. del módulo `ls_controller_server`, mostradas en la figura 8.7). Sin embargo, el componente LiveScheduler delega este comportamiento en un componente externo, el servidor de vídeo VoDKA [200], para poder así emitir los eventos programados por el usuario para componer las emisiones esperadas.

De esta forma, el componente LiveScheduler realmente no comienza ni detiene ninguna emisión por sí mismo, sino que interactúa con un componente externo para que lo haga. Este componente externo es el servidor de vídeo VoDKA, el cual permite programar la emisión de canales *multicast* usando diferentes tipos de contenidos: ficheros locales, fuentes HTTP, otras fuentes *multicast*, etc. Así, el componente LiveScheduler se integra con el componente VoDKA para la creación de canales *multicast* con diferentes contenidos, según la configuración de eventos y emisiones realizada por el administrador, simplemente teniendo la responsabilidad de invocar las operaciones oportunas en el componente VoDKA en el momento adecuado para comenzar o parar los eventos definidos (las cuales se corresponden con las operaciones `create_source`, `start_source`, `stop_source`, `start_output`, etc. que ofrece la clase `boxcodertcp_facade` del componente VoDKA, y que se muestran en la figura 8.7).

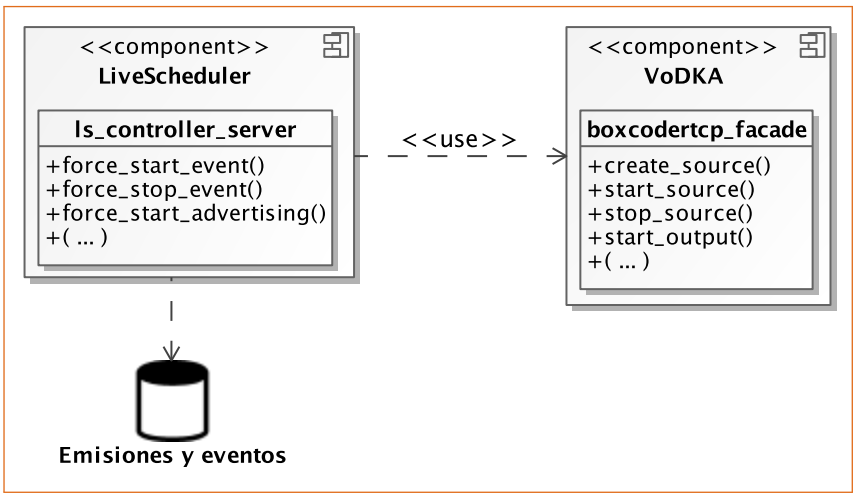


FIGURA 8.7: *Integración del componente LiveScheduler con el componente VoDKA*

Por tanto, la integración entre el componente LiveScheduler y el componente VoDKA es un factor clave en el buen funcionamiento del sistema completo. Aunque existen otros aspectos que pueden y deben ser probados aquí, como el acceso de LiveScheduler a su base de datos local de eventos y emisiones (creación, actualización y borrado de eventos y emisiones), o la interpretación del tiempo para que los eventos sean emitidos a la hora esperada; el funcionamiento global del sistema depende fundamentalmente de la integración con el componente VoDKA.

La implementación del componente LiveScheduler ha sido realizada durante el desarrollo de esta metodología de pruebas, por lo que su integración con el componente VoDKA se ha implementado teniendo en cuenta la aproximación aquí descrita, es decir, no existían pruebas previas que comprobasen el funcionamiento de dicha integración. En cualquier caso, usando la metodología de pruebas de integración propuesta en este capítulo, ha sido posible implementar y comprobar la integración entre estos dos componentes. De esta forma, aplicar esta metodología implica la creación de un componente *espía* que reemplace al componente VoDKA, interceptando y registrando las llamadas procedentes del componente LiveScheduler hacia este componente, y devolviendo datos que cumplan la API de integración del propio componente VoDKA. De esta forma, ni el componente LiveScheduler ni el componente VoDKA deben ser modificados para llevar a cabo estas pruebas de integración entre los mismos.

8.3.2.1. Construcción de la máquina de estados QuickCheck

De la misma forma que en el capítulo 6 la aproximación de pruebas propuesta consiste en utilizar una máquina de estados QuickCheck, bien escrita manualmente o bien generada a partir de una especificación en un lenguaje de modelado, para probar la API de integración que proporciona un componente; en esta ocasión, se

utilizará el mismo tipo de aproximación para invocar las operaciones que proporciona el componente LiveScheduler. La diferencia radica en que, en este caso, en las postcondiciones de dicha máquina de estados QuickCheck, el principal objetivo es comprobar que se han realizado las llamadas esperadas al componente externo VoDKA, y que se procesan de forma adecuada los datos devueltos por el mismo.

Los comandos que forman parte de la máquina de estados QuickCheck son, en este caso, las operaciones de la API de integración que proporciona el componente LiveScheduler que involucran interacciones con el componente externo VoDKA, es decir: comenzar a emitir un evento, detener la emisión un evento, comenzar a emitir un medio de relleno, detener la emisión de un medio de relleno, comenzar a emitir un medio de publicidad asociado a un evento, detener la emisión de publicidad, etc. Para mantener la simplicidad, se mostrarán únicamente dos operaciones: comenzar la emisión de un evento (`start_event`) y detener la emisión de un evento (`stop_event`), aunque para el resto de operaciones la estructura es exactamente la misma. Con esta información es posible implementar la función `command` que especifica las operaciones a ejecutar por la máquina de estados QuickCheck de la siguiente forma:

```
command(S) ->
 eqc_gen:oneof([{call, ?MODULE, start_event,
     [eqc_gen:oneof(S#state.events)]},
    {call, ?MODULE, stop_event,
     [eqc_gen:oneof(S#state.events)]}]).
```

Como se observa, tanto la operación `start_event` como `stop_event` reciben como argumento un elemento que se obtiene del estado de la máquina de estados QuickCheck. Dicho estado, además de la lista `events`, también contendrá otra lista, `current_events`:

```
-record(state, {events,
          current_events}).
```

Cabe recordar que el estado debe almacenar toda la información necesaria para poder realizar las comprobaciones oportunas en las postcondiciones (aunque también estará disponible en las precondiciones y función `next_state`). En este caso, cada elemento de la lista de eventos `events` es una tupla con dos elementos: `{EmissionId, Event}`, donde `Event` contiene los datos del evento, y `EmissionId` es el identificador de la emisión a la que el evento ha sido asignado. Por otro lado, `current_events` es una lista de registros `current_event`, que contienen la información sobre cuál es el evento actual para cada emisión. Esta es la definición del registro `current_event`:

```
-record(current_event, {emissionId,
                        event,
                        outputId}).
```

donde `emissionId` es el identificador de una emisión, `event` contiene los datos del evento actual para dicha emisión, y `outputId` es el identificador del canal definido en el componente externo VoDKA en el que el evento actual está siendo emitido. Además, `event` y `outputId` podrían tomar el valor `undefined` si no existe un evento actual para la emisión.

La inicialización del estado de partida para cada caso de prueba, realizado a través de la definición de la función `initial_state`, se implementa de la siguiente manera:

```
initial_state()->
 #state {events = get_all_events(),
         current_events = get_all_current_events()}.
```

La función `get_all_events` devuelve una lista de tuplas formada por los elementos `{EmissionId, Event}` que se obtienen del propio componente LiveScheduler después de realizar un proceso de inicialización que elimina las emisiones existentes y crea una lista inicial (usando la función `init_emissions` que se mostrará posteriormente en la función `prop_process`), que no cambiará durante toda la ejecución de la prueba. Por otro lado, con respecto al campo `current_events`, la función `get_all_current_events` devuelve una lista de registros `current_event`, representando una emisión y su evento actual para cada una. Como estos eventos estarán inicialmente detenidos, la lista devuelta por la función `get_all_current_events` serán registros cuyos campos `event` y `outputId` tendrán el valor `undefined`. Por ejemplo, si inicialmente se crean dos emisiones con identificadores `e1` y `e2`, este sería el resultado devuelto por la función `get_all_current_events`:

```
[
 #current_event {
  emissionId = "e1",
  event = undefined,
  outputId = undefined
 },
 #current_event {
  emissionId = "e2",
  event = undefined,
  outputId = undefined
 }
]
```

Los eventos almacenados en el campo `events` del estado serán utilizados como argumentos en las funciones `start_event` y `stop_event` usadas en la función `command`. Éstas son funciones envoltorio que se definen en el propio módulo de pruebas, y que llamarán a las operaciones reales del componente LiveScheduler para comenzar y detener la emisión de eventos (definidas en el módulo `ls_controller_server`), en concreto:

- `force_start_event(EmissionId, EventId)`, que comienza la emisión del evento con identificador `EventId` asociado a la emisión con identificador `EmissionId`.
- `force_stop_event(EmissionId, EventId)`, que detiene la emisión del evento con identificador `EventId` perteneciente a la emisión con identificador `EmissionId`.

Por tanto, las funciones envoltorio `start_event` y `stop_event` llamarán a las funciones `force_start_event` y `force_stop_event` respectivamente. Por ejemplo, esta es la implementación para la función `start_event`:

```
start_event({EmissionId, Event})->
 EventId = Event#event.eventId,
 EventInformation = ls_controller_server:force_start_event(
    EmissionId, EventId),
 OutputId = EventInformation#event_information.outputId,
 OutputId.
```

Como se observa, tanto los parámetros recibidos como la información devuelta por ambas funciones es distinta. Así, mientras que la función `start_event` recibe como parámetro una tupla `{EmissionId, Event}`, que es precisamente el tipo de dato que se almacena en el campo `events` del estado, la función `force_start_event` de LiveScheduler recibe dos parámetros, el identificador de la emisión `EmissionId` y el valor `EventId`, es decir, el identificador del evento `Event`. Con respecto al valor de retorno, la función `start_event` devuelve el elemento `outputId`, que se extrae del registro `event_information`, puesto que se necesitará en las postcondiciones para realizar ciertas comprobaciones. Este es el motivo por el que se usan este tipo de funciones envoltorio, puesto que, de esta manera, el código de la máquina de estados se mantiene limpio y sencillo, y además, se mantiene intacta la implementación del componente a probar (SUT) y del componente dependiente (DOC).

Una vez definidas las operaciones que se invocarán a la hora de ejecutar la máquina de estados, se deben definir las precondiciones y postcondiciones asociadas a cada una de ellas, así como la forma en la que modifican el estado.

Con respecto a la precondiciones, aunque LiveScheduler proporciona una interfaz de usuario que permite manejar emisiones y eventos, y esta interfaz no permite al

usuario invocar operaciones en estados incorrectos, como, por ejemplo, comenzar la emisión de un evento ya comenzada (puesto que únicamente se mostrará el botón de comenzar si el evento está parado), se ha tomado la decisión de que se pruebe qué sucede si ocurre una situación de este tipo. Esto es debido a que el uso de la API de integración que proporciona LiveScheduler no limita el uso de sus operaciones. Por tanto, cualquier operación podrá ser invocada en cualquier estado, por lo que las precondiciones siempre devolverán el valor `true`:

```
precondition(S, C) -> true.
```

Por otro lado, la lista de emisiones y eventos creados en el sistema, con sus estados correspondientes, los cuales se almacenan en el estado del módulo de pruebas, debe ser actualizada a medida que se ejecutan las diferentes operaciones de las pruebas. Para ello, se usa la función `next_state`. Por ejemplo, la cláusula de la función `next_state` correspondiente a la operación de comenzar la emisión de un evento, `start_event`, es la siguiente:

```
next_state(S, R, {call, ?MODULE, start_event,
   [{EmissionId, Event}]})->
 NewCurrentEvent = #current_event {
  emissionId = EmissionId,
  event = Event,
  outputId = R
 },
 NewCurrentEvents = replace_current_event(
    NewCurrentEvent, S#state.current_events),
 S#state {
  current_events = NewCurrentEvents
 };
```

Como se observa, la lista de eventos, `events`, no se modifica. Esto es debido a que simplemente se están comenzando y deteniendo eventos, pero no creando o borrando. Sin embargo, la lista `current_events` debe ser actualizada en función de la operación ejecutada. En este caso se debe reemplazar el evento actual de la emisión correspondiente con el nuevo evento a ser comenzado, lo cual se realiza usando la función `replace_current_event`. Evidentemente, para la operación `stop_event`, cuando el evento actual de una emisión se detiene, la lista `current_events` debe ser cambiada reemplazando el evento actual de una emisión por un evento `undefined`.

Por último, en las postcondiciones, se realizarán las acciones oportunas para comprobar si el componente se está comportando de la manera esperada. En este caso, el objetivo principal de estas pruebas es comprobar la integración entre LiveScheduler y el componente VoDKA, más que el comportamiento interno de LiveScheduler, el cual, evidentemente, también puede ser comprobado. Por tanto, antes de mostrar

en detalle el código de las postcondiciones, es necesario resolver cómo comprobar si LiveScheduler está realmente invocando las operaciones esperadas en el componente externo VoDKA, y usa los valores que éste devuelve de la manera adecuada. La siguiente sección (sección 8.3.2.2) describe cómo realizar esta tarea.

8.3.2.2. Uso del componente *espía*

Usar un componente de reemplazo para realizar las pruebas de integración en este caso de estudio significa substituir el componente VoDKA por otro componente, en este caso, un componente *espía*. Este componente debe ofrecer la misma API de integración usada por el componente LiveScheduler que el propio componente VoDKA ofrece para que, de esta forma, ni el componente LiveScheduler ni el componente VoDKA deban ser modificados para llevar a cabo este tipo de pruebas.

La figura 8.8 representa la arquitectura de pruebas en este caso. Esta figura muestra los componentes utilizados en el caso concreto de LiveScheduler siguiendo la arquitectura genérica mostrada en la figura 8.3. En este caso, el componente `boxcodertcp_operations` representa el componente que registra las trazas de invocación al componente externo VoDKA, y `boxcodertcp_facade` es el componente que reemplaza la parte del componente VoDKA usada por LiveScheduler.

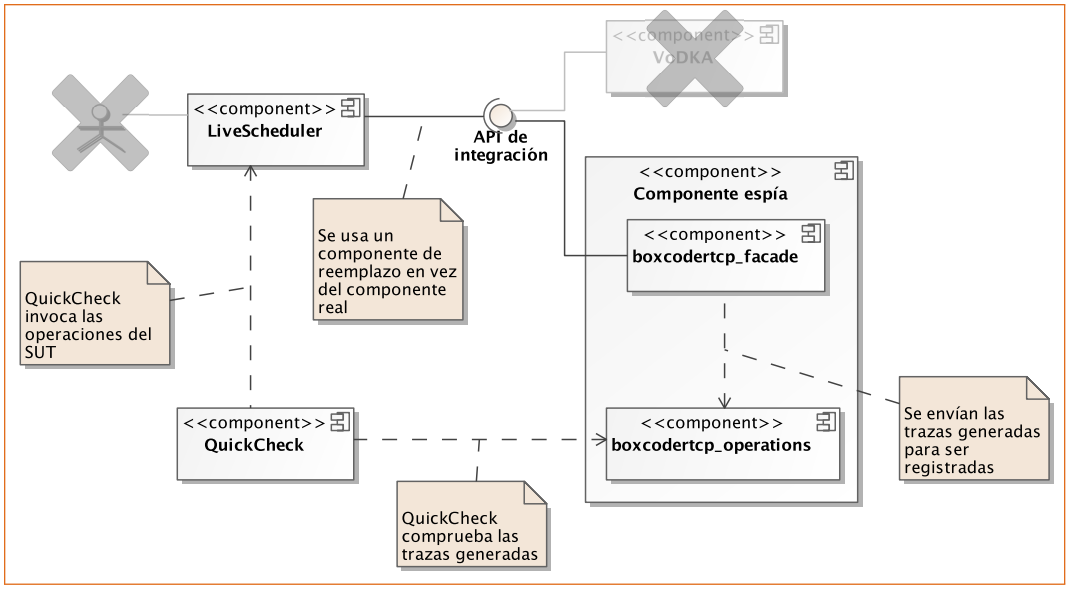


FIGURA 8.8: *Arquitectura propuesta para probar la integración del componente LiveScheduler con el componente VoDKA usando un componente de reemplazo*

Este componente de reemplazo no debe replicar la funcionalidad ofrecida por el componente VoDKA (sería replicar la implementación de este componente), sino que puede simularla. Así, por ejemplo, en este caso, este componente no comenzará

ni detendrá la emisión de ningún evento, pero para el componente LiveScheduler actuará como si lo hiciese. De esta forma, el componente LiveScheduler no notará ningún tipo de diferencia entre el componente original y el componente de reemplazo.

En cualquier caso, una de las partes más importantes del componente *espía* es el registro de accesos a las operaciones del componente reemplazado, en este caso, del componente VoDKA, para comprobar, posteriormente, si éstas han sido invocadas por el componente LiveScheduler. En este caso, el componente Erlang `boxcodertcp_operations` ofrece las siguientes operaciones:

- `addOperation(Operation)`: añade la operación `Operation` a la lista de operaciones invocadas, donde una operación se define como una tupla con los siguientes elementos `{Module, Function, Arguments, Result}`, siendo `Module` el nombre del módulo que contiene la operación, `Function` el nombre de la operación invocada, `Arguments` la lista de argumentos usados en la invocación y `Result` el resultado devuelto por la operación.

- `getAllOperations()`: devuelve la lista de operaciones invocadas e inicializa dicha lista.

Así, cada vez que se invoca una operación del componente *espía*, este acceso se registra a través del módulo `boxcodertcp_operations`. Por ejemplo, la operación `create_source`, perteneciente a la API de integración proporcionada por el componente VoDKA, es una de las operaciones usada por la función `force_start_event` de LiveScheduler para comenzar a emitir un evento, por lo que debe ser implementada en el componente de reemplazo `boxcodertcp_facade`. La implementación que se ha realizado es la siguiente:

```
create_source(Session, Source)->
 R = gen_server:call(Session, {create_source, Source}),
 boxcodertcp_operations:addOperation(
    {boxcodertcp_facade, create_source, [Source], R}),
 R.
```

Para simular el comportamiento de la implementación real del componente VoDKA, el componente de reemplazo usa un servidor genérico (`gen_server` [23]) en el que, en este ejemplo concreto, almacena las fuentes creadas por el componente LiveScheduler. De esta forma, si un componente crea una fuente y, posteriormente, obtiene la lista de fuentes creadas, la nueva fuente será devuelta como una de ellas. De esta forma, aunque esta implementación no realiza acciones reales, para otros componentes externos parece que realmente sí que las está ejecutando. Esta es la implementación que almacena las fuentes creadas en el estado del servidor genérico:

```
handle_call({create_source, Source}, _From, State) ->
 SourceId = State#state.id,
 NewSource =
    Source#source {
        source_id = SourceId
      },
 NewState =
    State#state {
     sources = [NewSource | State#state.sources],
     id = SourceId + 1
    },
 {reply, {ok, SourceId}, NewState};
```

Una vez aclarado cómo funciona el componente *espía*, éste debe ser usado en el módulo de pruebas. En concreto, en las postcondiciones de cada operación se debe comprobar si se han invocado las operaciones esperadas del componente dependiente. Para ello, la solución propuesta consiste en devolver las operaciones invocadas en las funciones envoltorio que representan las transiciones de la máquina de estados QuickCheck. De esta forma, la implementación de la función envoltorio para la operación `start_event` se modificaría de la siguiente forma:

```
start_event({EmissionId, Event})->
 EventId = Event#event.eventId,
 EventInformation = ls_controller_server:force_start_event(
    EmissionId, EventId),
 Operations = boxcodertcp_operations:getAllOperations(),
 OutputId = EventInformation#event_information.outputId,
 {Operations, OutputId}.
```

Así, después de invocar la función correspondiente, en este caso, la función `force_start_event` del módulo `ls_controller_server` de LiveScheduler, se obtiene la lista de operaciones invocadas en el componente VoDKA junto con sus valores de retorno, utilizando para ello la función `getAllOperations` definida en el módulo `boxcodertcp_operations`. Esta traza debe ser inspeccionada para comprobar que se han realizado las llamadas esperadas al componente VoDKA al ejecutar esta operación. Puesto que este tipo de comprobaciones no deben formar parte de la función envoltorio, sino que se deben realizar en las postcondiciones, la traza de operaciones (`Operations`) se devuelve junto con el valor de retorno `OutputId` como resultado de esta función (que será usado para almacenar en el estado como un valor simbólico y realizar comprobaciones adicionales).

Finalmente, la postcondición de la operación `start_event` recibirá, además de los argumentos con los que la función envoltorio `start_event` fue invocada y la información almacenada en el estado, el resultado que devuelve dicha función envoltorio, el cual contiene la traza de operaciones invocada en el componente

VoDKA al ejecutar dicha función. Con esta información es posible comprobar, en la implementación de la postcondición, si la lista de operaciones invocadas en el componente VoDKA es la esperada:

```
postcondition(S, {call, ?MODULE, start_event,
    [{EmissionId, Event}]}, R)->
 Operations = erlang:element(1, R),
 check_start_event(S#state.current_events, EmissionId, Event,
    Operations);
```

La función auxiliar `check_start_event` construye la lista de operaciones esperada que se llaman para comenzar la emisión de un evento, y las compara con las operaciones realmente ejecutadas. Se debe recordar que la lista de operaciones almacenadas contienen tanto los argumentos con los que se invoca cada una de ellas, como el resultado que devuelven, puesto que la lista de operaciones a ejecutar puede depender del resultado de otras operaciones ya ejecutadas. En cualquier caso, el resultado de esta comparación será el resultado de la postcondición. La figura 8.9 muestra un ejemplo de las interacciones entre los diferentes componentes que ocurren durante este proceso. Se debe tener en cuenta que esta aproximación se seguirá para todas las operaciones a probar.

8.3.2.3. Generación y ejecución de los casos de prueba

Una vez construida la máquina de estados es posible ejecutarla usando la propia herramienta QuickCheck. Para ello, se debe escribir la propiedad que generará secuencias de operaciones con los comandos especificados, la cual tendrá en cuenta las precondiciones y comprobará las postcondiciones después de cada ejecución:

```
prop_process() ->
 delete_all_emissions(),
 init_emissions(),
 ?FORALL(Cmds, eqc_statem:commands(?MODULE),
  begin
   {_H, _S, Res} = eqc_statem:run_commands(?MODULE, Cmds),
   stop_all_emissions(),
   eqc_statem:pretty_commands(?MODULE, Cmds, {H, S, Res},
      Res == ok)
 end).
```

En esta propiedad se puede observar que antes de ejecutar las pruebas se inicializa el sistema eliminando todas las emisiones almacenadas en base de datos, utilizando la función `delete_all_emissions`, y creando un conjunto específico de emisiones y eventos a través de la función `init_emissions`. Por otro lado, después de ejecutar cada iteración, todos los eventos creados se devuelven a su estado inicial

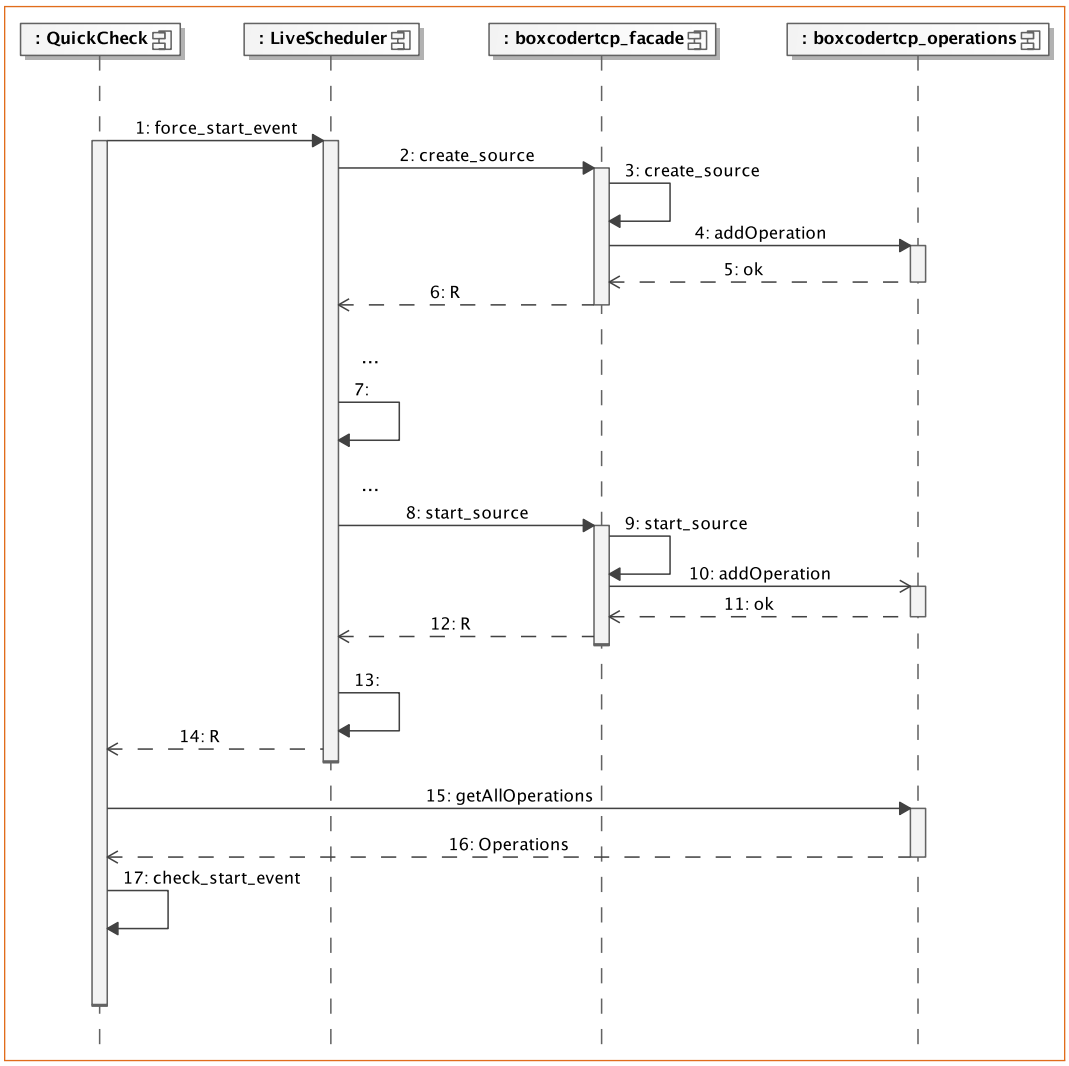


FIGURA 8.9: *Ejemplo de interacciones entre QuickCheck, LiveScheduler y el componente de reemplazo en las pruebas de integración*

llamando a la función `stop_all_emissions`, la cual detiene la emisión de todos los eventos, sin borrar las emisiones y eventos existentes, ya que, en este caso, siempre se usarán las mismas emisiones y eventos creados a través de la función `init_emissions`.

Gracias a la aleatoriedad introducida por los generadores de datos, serán ejecutadas diferentes combinaciones de secuencias de operaciones, muchas de ellas correspondientes a secuencias típicas de operaciones que el usuario ejecutará usando la interfaz de usuario, y otras más inusuales e improbables, incrementando, de esta forma, las posibilidades de encontrar un defecto en el software. Un ejemplo de secuencia de operaciones generada aleatoriamente por QuickCheck podría ser la

siguiente (evidentemente, al introducir más comandos posibles en la especificación se obtendrían más combinaciones, usuales e inusuales, de operaciones):

```
{call, ls_eqc_process, start_event, [{"28" {event, "Ev4", ...}}]}
{call, ls_eqc_process, start_event, [{"28" {event, "Ev4", ..}}]}
{call, ls_eqc_process, stop_event, [{"26" {event, "Ev2", ...}}]}
{call, ls_eqc_process, start_event, [{"26" {event, "Ev1", ...}}]}
{call, ls_eqc_process, stop_event, [{"28" {event, "Ev4", ...}}]}
{call, ls_eqc_process, start_event, [{"28" {event, "Ev1", ...}}]}
{call, ls_eqc_process, stop_event, [{"28" {event, "Ev1", ...}}]}
{call, ls_eqc_process, stop_event, [{"28" {event, "Ev1", ...}}]}
...
```

De esta forma, al ejecutar esta traza de operaciones se invocarán las funciones envoltorio que llamarán a las operaciones del componente LiveScheduler, las cuales, a su vez, realizarán llamadas a las operaciones correspondientes en el componente externo VoDKA. Por ejemplo, al ejecutar la transición `start_event` con el evento con identificador `Ev4` perteneciente a la emisión `28`, se ejecuta la función envoltorio `start_event`, la cual finalmente llamará a la función `force_start_event` de LiveScheduler. Esta función delegará algunas operaciones en el componente externo VoDKA, el cual cabe recordar que ha sido reemplazado por otro componente (`boxcodertcp_facade`) que registra las operaciones llamadas (usando el módulo `boxcodertcp_operations`).

Un ejemplo de traza almacenada por el componente *espía* al ejecutar la función `force_start_event` de LiveScheduler es la siguiente:

```
[{boxcodertcp_facade,status,[0,1311],running},
 {boxcodertcp_facade,stop_output,[1,1315],ok},
 {boxcodertcp_facade,delete_output,[1,1315],ok},
 {boxcodertcp_facade,get_outputs,[1],[]},
 {boxcodertcp_facade,stop_source,[1],ok},
 {boxcodertcp_facade,stop_source,[0],ok},
 {boxcodertcp_facade,get_outputs,
              [0],
              [{udp_output,1311,
                       "Ev4",
                       "localhost",
                       0,
                       0,
                       15000000,
                       10,
                       [],
                       true}]},
```

```
{boxcodertcp_facade,add_output,
              [0,
               {udp_output,undefined,
                       "Ev1",
                       "239.194.1.1",
                       11411,
                       0,
                       15000000,
                       10,
                       [],
                       true}],
              1316},
{boxcodertcp_facade,start_source,[0],ok}]
```

Esta traza representa la secuencia de todas las funciones invocadas en el componente externo VoDKA. Así, en este ejemplo, la primera llamada fue la función `status` del módulo `boxcodertcp_facade` con argumentos `(0, 1311)`, que devolvió el átomo `running`; después se invocó la función `stop_output` con los argumentos `(1, 1315)` y devolviendo el átomo `ok`, etc. Esta traza de operaciones puede ser recuperada a través del módulo `boxcodertcp_operations`, y así, poder ser comparada con la traza de operaciones esperada teniendo en cuenta los argumentos y valores de retorno. Para realizar esta comprobación, en el caso de la operación `start_event`, se usa la función `check_start_event`.

Dicha comparación se usará para determinar si la postcondición se cumple o no. Se debe tener en cuenta que, si se produce un fallo en la ejecución de las pruebas, es decir, si la postcondición no se cumple, la herramienta QuickCheck ejecutará un proceso de reducción que, a partir de una traza de operaciones que ha fallado, permitirá reducirla hasta encontrar una traza de operaciones más pequeña que también falla [87, 135, 374].

En resumen, la máquina de estados QuickCheck generará secuencias aleatorias de operaciones, como comenzar o detener la emisión de un evento. Cada vez que una de estas operaciones se ejecuta, el componente LiveScheduler llama a una o varias operaciones en el componente externo VoDKA, en este caso, en el componente de reemplazo. Este componente de reemplazo responderá de la misma manera que lo haría el componente original VoDKA, a la vez que registra su actividad almacenando la traza de operaciones invocada y sus valores de retorno. Cuando se comprueba la postcondición para cada operación, se recupera esta traza de operaciones y se compara con la esperada. Si ambas no coinciden, QuickCheck reportará un error, junto con la secuencia de operaciones que ha producido el error y su versión minimizada. En caso contrario, se ejecutará la siguiente secuencia de operaciones hasta completar la ejecución de todas las secuencias generadas.

8.3.2.4. Pruebas de integración negativas: control de errores en la comunicación

Hasta ahora, el tipo de pruebas de integración realizadas con el caso de estudio del componente LiveScheduler permiten comprobar la integración entre el propio componente LiveScheduler y el componente externo VoDKA cuando no hay fallos en dicha comunicación. Sin embargo, aunque la integración con el componente externo VoDKA ha sido probada, cuando LiveScheduler se utiliza en un entorno de producción, es posible obtener errores producidos por fallos en la comunicación entre ambos componentes. Por tanto, es interesante comprobar que el componente LiveScheduler se comporta de la manera esperada cuando la comunicación entre los dos componentes no se comporta de la forma apropiada, por ejemplo, cuando hay congestión en la red, o si hay errores de comunicación como *timeouts* (peticiones no atendidas a tiempo). Las pruebas de integración negativas permiten comprobar este tipo de situaciones.

En el caso concreto de la integración entre el componente LiveScheduler y el componente VoDKA, este tipo de fallos pueden ocurrir de manera frecuente, puesto que el componente VoDKA realiza operaciones complejas que potencialmente pueden fallar dependiendo del estado de la red. Por tanto, si no se considera esta situación en las pruebas, no será posible comprobar cómo estos fallos influyen en el comportamiento de LiveScheduler.

Siguiendo la misma aproximación descrita hasta ahora, realizar este tipo de pruebas no debería suponer la modificación de ninguno de los componentes (ni LiveScheduler ni VoDKA), por lo que se debería usar el componente *espía* para simular esta situación. Por ejemplo, una posible forma de producir peticiones no atendidas a tiempo es provocar que el componente de reemplazo no responda a las peticiones inmediatamente, sino que espere varios segundos para obtener el resultado de invocar una función.

El siguiente aspecto a tratar es cómo decidir si el componente dependiente debe responder inmediatamente o después de un período de tiempo. Además, esta información la debe conocer el módulo de pruebas, es decir, el módulo de pruebas necesita saber de antemano si una operación va a ser atendida inmediatamente por el componente de reemplazo o va a tardar, puesto que la traza de operaciones final a ser invocada en cada caso variará. Una solución para llevar a cabo esta aproximación es crear una nueva transición en la máquina de estados QuickCheck que cause un retraso en las respuestas del componente de reemplazo. Además, es posible usar la función `frequency` proporcionada por la propia herramienta QuickCheck para establecer una frecuencia diferente para cada transición de la máquina de estados. Así, es posible simular redes en las que los retrasos en las respuestas sean mayores y más frecuentes, y redes menos saturadas en las que estos retrasos sean menores y menos frecuentes. Por ejemplo, la función `command` podría ser reescrita de la siguiente manera:

```
command(S) ->
 eqc_gen:frequency(
    [{1, {call, ?MODULE, delay,
       [?LET(MSec, eqc_gen:choose(1, 10000),
          (10000 - MSec))]}},
     {10, eqc_gen:oneof([
        {call, ?MODULE, start_event,
           [eqc_gen:oneof(S#state.events)]},
        {call, ?MODULE, stop_event,
           [eqc_gen:oneof(S#state.events)]}])
     }
    ]).
```

Esta implementación implica que, una de cada diez veces, se producirá un retraso en el componente de reemplazo. Para la transición `delay`, se usa el generador `choose`, que forma parte de los generadores proporcionados por QuickCheck para generar un número aleatorio en un rango específico, en este caso, entre 1 y 10000, que representa los milisegundos que se usarán como retraso.

Por su parte, la implementación de la función `delay` en el módulo de pruebas debe indicar al componente de reemplazo cuánto tiempo debe esperar para responder la siguiente petición. Esta delegación se realiza enviando un mensaje directo al servidor genérico `boxcodertcp_facade`:

```
delay(MSec) ->
 Session = boxcodertcp_session:get_session(),
 boxcodertcp_facade:delay(Session, MSec).
```

donde `boxcodertcp_session` es un nuevo servidor genérico Erlang (es decir, `gen_server`) en el que el módulo `boxcodertcp_facade` registrará su identificador de proceso (esto es, su `pid`), puesto que, de esta forma, será posible consultarlo en cualquier momento. Esto se realiza porque este valor es necesario conocerlo para invocar las operaciones del módulo `boxcodertcp_facade` (como `create_source`, `start_source`, `stop_source`, `start_output`, o la propia función `delay`).

Como se observa, se ha creado una nueva comunicación en el escenario original, mostrado en la figura 8.10, entre el módulo QuickCheck y el componente de reemplazo. Además, se ha añadido una nueva operación al componente de reemplazo `boxcodertcp_facade` llamada `delay`, la cual no existe en el componente externo original VoDKA. Esta operación almacena en el estado del servidor genérico el tiempo de respuesta de la siguiente petición:

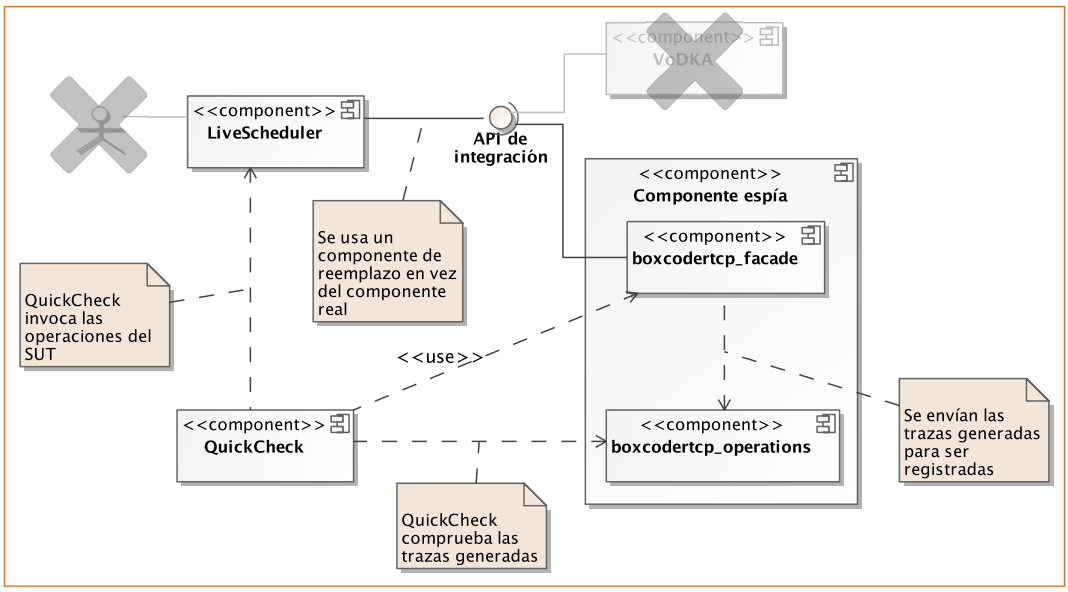


FIGURA 8.10: *Arquitectura propuesta para realizar las pruebas de integración negativas del componente LiveScheduler con el componente VoDKA*

```
delay(Session, MSec)->
 gen_server:call(Session, {set_delay, MSec}).

handle_call({set_delay, MSec}, _From, State) ->
 NewState = State#state {
  delay = MSec
 },
 {reply, ok, NewState};
```

De esta forma, cada operación implementada en este componente debe leer este tiempo y tenerlo en cuenta para generar las respuestas. Por ejemplo, la implementación de la función `create_source`, mostrada anteriormente, se cambia por el siguiente código:

```
create_source(Session, Source)->
 generate_timeout(Session, {gen_server, call,
    [Session, {create_source, Source}]}),
 R = gen_server:call(Session, {create_source, Source}),
 boxcodertcp_operations:addOperation(
    {boxcodertcp_facade, create_source, [Source], R}),
 R.
```

donde `generate_timeout` es una función que retrasa la ejecución de una tarea, o produce un *timeout* si el tiempo de espera es demasiado largo:

```
generate_timeout(Session, Op)->
 MSec = gen_server:call(Session, get_delay),
 if
  MSec > 5000 ->
   exit({timeout, Op});
  true ->
   timer:sleep(MSec)
 end.
```

La implementación de la función interna que obtiene el valor del retraso a aplicar y, posteriormente, lo inicializa al valor por defecto (0 milisegundos), es la siguiente:

```
handle_call(get_delay, _From, State) ->
 MSec = State#state.delay,
 NewState = State#state {
  delay = 0
 },
 {reply, MSec, NewState};
```

Por otro lado, es necesario modificar la estructura del estado de la máquina de estados QuickCheck, puesto que ahora es necesario conocer el tiempo de respuesta de la siguiente operación. Para ello, se añade un nuevo campo `time_response` a la información almacenada en el estado:

```
-record(state, {events,
          current_events,
          time_response}).
```

Este nuevo campo se rellena en la función `next_state` para la nueva transición `delay`:

```
next_state(S, _R, {call, ?MODULE, delay, [MSec]})->
 S#state {
  time_response = MSec
 }.
```

Evidentemente, el resto de campos no quedan afectados por la ejecución de esta operación.

De esta forma, antes de ejecutar el resto de operaciones de la máquina de estados, como, por ejemplo, `start_event` o `stop_event`, es posible conocer el tiempo de respuesta del componente externo. Después de ejecutar estas operaciones, este tiempo será inicializado hasta la próxima ejecución de otra transición `delay`. Esta inicialización se realiza en la función `next_state`, por ejemplo, para la función `start_event`:

```
next_state(S, R, {call, ?MODULE, start_event,
   [{EmissionId, Event}]})->
 NewCurrentEvent = #current_event {
  emissionId = EmissionId,
  event = Event,
  outputId = {call, erlang, element, [2, R]}
 },
 NewCurrentEvents = replace_current_event(
    NewCurrentEvent, S#state.current_events),
 S#state {
  current_events = NewCurrentEvents
  time_response = 0
 };
```

Con respecto a las precondiciones, se considera que la operación `delay` no debe ser ejecutada dos veces de forma consecutiva, es decir, si un valor ya ha sido asignado para el siguiente tiempo de respuesta, no es necesario establecer uno nuevo:

```
precondition(S, {call, ?MODULE, delay, [_MSec]})->
 S#state.time_response == 0;
```

Para las postcondiciones, no se considera interesante comprobar ningún tipo de condición, por lo que, la función `postcondition` devolverá el valor `true` para este caso:

```
postcondition(_S, {call, ?MODULE, delay, [_MSec]}, _R)->
 true;
```

Ejecutando las pruebas con esta nueva transición, es decir, generando tiempos de espera y excepciones en la comunicación entre los dos componentes, es posible comprobar el comportamiento de LiveScheduler si la comunicación con el componente VoDKA es lenta o incluso se producen errores como peticiones no atendidas a tiempo. Por ejemplo, la figura 8.11 muestra un ejemplo de ejecución en el que, inicialmente, el tiempo de espera es 0, por lo que cualquier comando que se ejecute en este estado devolverá su resultado inmediatamente, como es el caso de la primera operación `start_event` mostrada en dicha figura. Sin embargo, si se ejecuta la operación `delay`, se almacena en el estado el tiempo de espera de la siguiente operación. Así, en el ejemplo de la figura 8.11, la segunda llamada a `start_event` tardará el tiempo `Msec` almacenado en el estado en devolver su respuesta (o se lanzará una excepción si este tiempo es mayor a 5 segundos).

En este punto, si se ejecutan de nuevo las pruebas, éstas fallan debido a que la traza esperada no es la misma a la generada cuando se produce una excepción. Por tanto, es importante conocer en las postcondiciones los tiempos de respuesta usados por el componente externo, el cual puede ser consultado a partir del campo

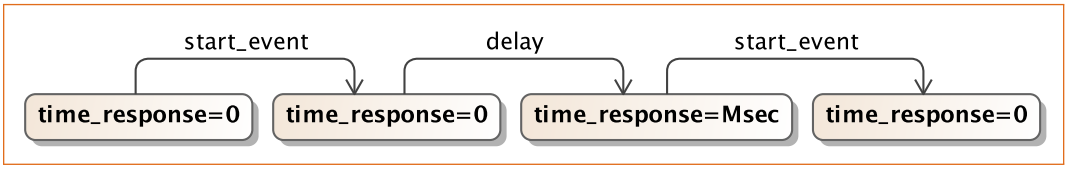


FIGURA 8.11: *Ejemplo de secuencia de operaciones en las pruebas de integración negativas del componente LiveScheduler*

`time_response` del estado. De esta forma, se pueden comparar las trazas, por ejemplo, en la función `check_start_event` (la cual pasaría a tener un nuevo parámetro que es el tiempo de respuesta), teniendo en cuenta dichos tiempos.

Además, se debe tener en cuenta que con esta aproximación se podrían incluso considerar situaciones más precisas. Por ejemplo, en este caso, el tiempo de espera es un parámetro global que afecta a todas las operaciones del componente externo, pero podría definirse un tiempo de respuesta independiente para cada operación del componente externo, y comprobar así el comportamiento del sistema a probar.

8.3.2.5. Integración de los dos componentes reales

La aproximación de pruebas descrita, en la que se usa un componente de reemplazo, permite también usar los dos componentes reales con un pequeño cambio en el componente *espía*. En concreto, la idea es transformar el componente *espía* en un *proxy* que redirija las llamadas al componente original. Para ello, en el caso de estudio de LiveScheduler, como se muestra en la figura 8.12, se debe cambiar la implementación del módulo `boxcodertcp_facade` para que, en vez de devolver datos de prueba, se invoque el componente real VoDKA para así devolver datos reales.

Por ejemplo, esta es la implementación de la función `create_source` en el módulo `boxcodertcp_facade`, la cual, además de soportar el retraso en la ejecución de operaciones y registrar que la operación ha sido llamada, invoca al componente real VoDKA para, posteriormente, devolver el resultado devuelto:

```
create_source(Session, Source)->
 generate_timeout(Session, {gen_server, call,
    [Session, {create_source, Source}]}),
 VodkaModule = get_vodka_module(),
 R = VodkaModule:create_source(Session, Source),
 boxcodertcp_operations:addOperation(
    {boxcodertcp_facade, create_source, [Source], R}),
 R.
```

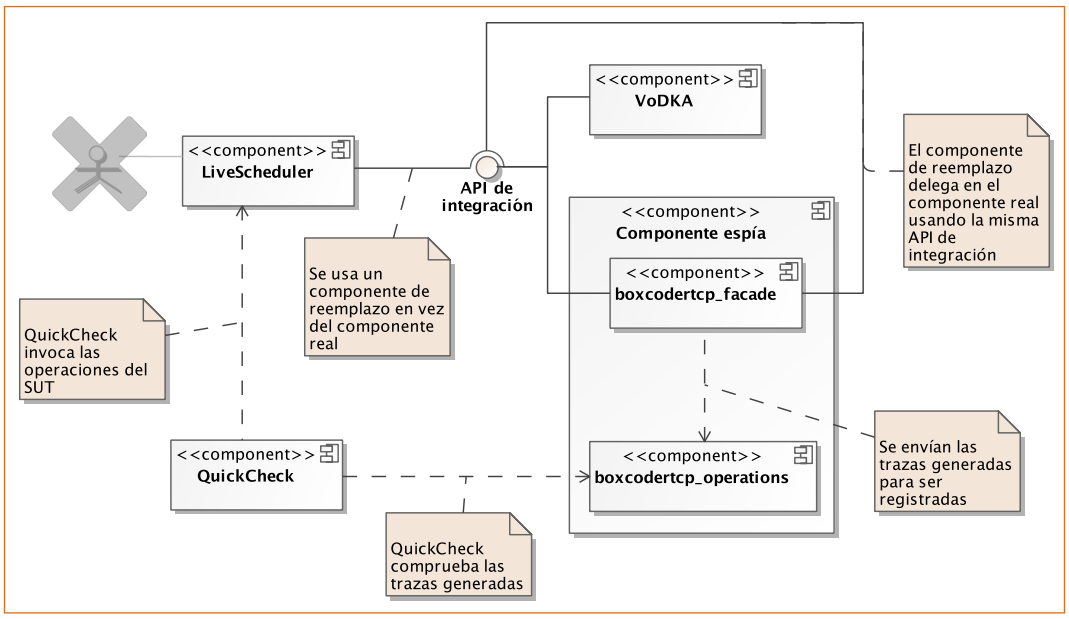


FIGURA 8.12: *Arquitectura propuesta para probar la integración del componente LiveScheduler con el componente VoDKA usando el propio componente real VoDKA*

donde `get_vodka_module` es una función que devuelve la interfaz de acceso al componente real VoDKA.

En este caso, las pruebas de integración sí que estarían utilizando los dos componentes reales, por lo que sería posible detectar defectos en la comunicación no encontrados en las pruebas realizadas hasta ahora. La ventaja de esta aproximación es que tanto el módulo de pruebas, como los componentes LiveScheduler y VoDKA quedarían intactos, es decir, no deben ser modificados para realizar este tipo de pruebas, sino que el código usado para probar la integración usando un componente de reemplazo sigue siendo totalmente válido en este caso, tanto para pruebas positivas como negativas.

8.3.3. Resultados de aplicar la metodología de pruebas

El uso de una herramienta como QuickCheck evita la tediosa y compleja tarea de diseñar y escribir grandes conjuntos de casos de prueba. Así, el modelo QuickCheck permite expresar, de una manera declarativa, las operaciones que forman parte de las pruebas, así como las precondiciones y postcondiciones de cada una de ellas. Por tanto, junto con la generación automática de datos de entrada para ejecutar cada una de las operaciones disponibles, QuickCheck proporciona un mecanismo para generar secuencias de operaciones utilizando una máquina de estados. De esta forma, en este caso, la ejecución de las pruebas comprueba diferentes secuencias

de operaciones, generadas automáticamente a partir de un conjunto de operaciones disponibles.

En esta ocasión, al ser LiveScheduler un componente en implementación cuando está metodología de pruebas se ha desarrollado, llevar a cabo estas pruebas ha permitido realizar la implementación de dicho componente generando una gran confianza en la integración del mismo con el componente externo VoDKA, puesto que se ha comprobado que éste invoca las operaciones esperadas del componente VoDKA en el momento y orden esperados. Así, por ejemplo, cuando LiveScheduler intenta comenzar la emisión de un evento, éste llama a las funciones apropiadas en el componente externo VoDKA y reacciona de la manera deseada ante los valores de retorno devueltos por este componente externo. Aunque no se ha mostrado en este capítulo, además de este tipo de pruebas de integración, durante la implementación de LiveScheduler también que se han creado modelos QuickCheck que permitían probar el comportamiento de todas las operaciones que ofrece.

Cabe mencionar que, una vez realizada la implementación de las pruebas de integración positivas con un resultado satisfactorio, el componente LiveScheduler fue usado en un entorno de preproducción para programar ciertos eventos y grabaciones. Desafortunadamente, dicho despliegue no funcionó de la manera esperada, apareciendo fallos en la comunicación del componente LiveScheduler y el componente VoDKA. Este fue el motivo que propició replantearse qué pruebas se habían realizado y por qué la comunicación entre ambos componentes falla si existe un modelo QuickCheck que indica lo contrario. En este punto, se llegó a la conclusión de que eran necesarias una serie de pruebas negativas que comprobasen que LiveScheduler se comportaba de la manera esperada cuando se producen fallos en la comunicación con el componente VoDKA.

Por tanto, con respecto a las pruebas negativas, se ha probado que LiveScheduler se comporta de la manera esperada cuando la comunicación con el componente VoDKA es lenta o se producen *timeouts*, es decir, no se producen respuestas de una manera inmediata. Durante este nuevo proceso de pruebas aparecieron fallos que no se produjeron durante la ejecución de las pruebas positivas, mayormente debidos a excepciones no capturadas. Además, al usar QuickCheck como herramienta de pruebas, se obtuvo información útil cuando se producían fallos con la ejecución de las pruebas. En concreto, cuando QuickCheck encuentra un error, éste devuelve la secuencia exacta de operaciones que causó este error, además de una secuencia mínima de operaciones equivalente que produce también un error [87, 135, 374]. En comparación con aquellas situaciones en las que únicamente se dispone de una breve descripción de un comportamiento extraño dada por el usuario acerca de un fallo encontrado en un sistema instalado en un entorno real (como ocurrió cuando LiveScheduler fue instalado en un entorno de preproducción después de ejecutar las pruebas positivas), estas trazas de operaciones son extremadamente útiles para realmente identificar y localizar el defecto en el código fuente antes de desplegar el

sistema software, o una nueva versión del mismo.

Una vez realizadas estas pruebas, solucionados los defectos que han aparecido durante las mismas, e instalado LiveScheduler de nuevo en el mismo entorno, no se han encontrado más fallos en el funcionamiento del mismo. Obviamente, la aproximación explicada para la realización de pruebas de integración negativas con el objetivo de comprobar este tipo de situaciones puede ser extendida para probar cualquier otro tipo de excepciones o errores que puedan ocurrir en la comunicación entre dos componentes, o incluso ante la ausencia total de respuestas.

Es importante recalcar que a partir de la versión v1.28.2 de QuickCheck, posterior al desarrollo de esta metodología, éste ya soporta de forma implícita el uso de componentes *mock*. De esta forma, usando esta nueva características de QuickCheck, se facilita el registro de trazas invocadas por cada operación, y se automatiza la comparación de dichas trazas invocadas con las trazas esperadas, las cuales se definen para cada una de las operaciones a probar. Por tanto, aunque la idea de la aproximación es exactamente la misma que la explicada aquí, su implementación con QuickCheck se simplifica.

8.4. Resumen

Este capítulo describe una aproximación metodológica para probar la integración entre componentes software. Este escenario es muy común en sistemas modernos, los cuales suelen estar estructurados en diferentes componentes que se integran entre sí para ofrecer un conjunto de servicios o proporcionar ciertas funcionalidades. En todas estas situaciones, la integración es un aspecto esencial a probar.

Esta metodología está basada en el uso de propiedades, las cuales se usan para generar casos de prueba que permiten comprobar la interacción entre dos componentes. Para ello, se ha usado un caso de estudio real que ilustra cómo se aplica esta aproximación usando QuickCheck como herramienta de pruebas para definir propiedades, en concreto, a través del uso de una máquina de estados. El uso de una máquina de estados QuickCheck permite definir de una manera declarativa las operaciones que formarán parte de las pruebas para que, a partir de éstas, se generen cientos o miles de secuencias de operaciones que conformarán los casos de prueba. La forma en la que se definen las transiciones de la máquina de estados, especificando precondiciones, postcondiciones y cambios de estado, permite a QuickCheck generar no sólo secuencias de operaciones que normalmente serían ejecutadas con el uso del componente, sino todas las secuencias que se ajusten al conjunto de restricciones especificadas en la definición de la máquina de estados, sin importar lo extrañas o improbables que puedan ser. De esta forma, se aumenta la posibilidad de encontrar defectos antes que el componente sea desplegado en un entorno de producción, lo cual potencialmente permite ahorrar tiempo y esfuerzo.

Por otro lado, el criterio para determinar si un caso de prueba ha sido ejecutado de forma satisfactoria está basado en reemplazar componentes externos dependientes por otros con una API de integración equivalente, además de inspeccionar las trazas de operaciones invocadas en la comunicación con el mismo. Naturalmente, el componente de reemplazo no necesita replicar el funcionamiento del componente original, únicamente debe proporcionar la misma interfaz y simular las respuestas, por lo que el tiempo de desarrollo del mismo no debería ser significativo.

Reemplazar componentes por otros más simples con una API de integración equivalente también tiene sus ventajas. Así, aunque se está probando la interacción entre dos componentes, es posible evitar efectos colaterales que pueden ocurrir si se usa el componente real. Además, es posible realizar estas pruebas incluso antes de que el componente con el que se integra el componente a probar esté finalizado. Esto es posible puesto que el aspecto clave de esta técnica es la inspección de las trazas de comunicación entre los dos componentes.

Por último, aunque esta aproximación se ha explicado usando la herramienta QuickCheck en combinación con un componente Erlang, este procedimiento de pruebas no está ligado únicamente al uso con esta herramienta, sino que puede ser fácilmente usado con otros lenguajes de programación, en otros entornos de integración, o con otras herramientas de pruebas basadas en propiedades que permitan llevar a cabo las técnicas explicadas en este capítulo. De la misma forma, el escenario particular mostrado con el caso de estudio LiveScheduler es muy habitual en sistemas software modernos, puesto que normalmente están estructurados en diferentes componentes software que funcionan juntos para ofrecer un servicio o una funcionalidad. En todas estas situaciones, la integración es un aspecto esencial a probar. Por tanto, al igual que se ha usado con el caso de estudio del programador de eventos LiveScheduler, esta metodología puede ser aplicada en otros contextos y casos similares, para probar, de manera genérica, la integración entre diferentes componentes software.

9

Evaluación de rendimiento en sistemas software integrados

9.1. Introducción

Una vez que los componentes software, implementados y probados de manera unitaria, se han integrado, es posible realizar las pruebas a nivel de sistema. Estas pruebas, denominadas pruebas de sistema, tratan el sistema software como un todo, y comprueban si el comportamiento del mismo es el esperado según los requisitos definidos. Dependiendo de si los requisitos comprobados son requisitos funcionales o no funcionales, estas pruebas se denominan, respectivamente, pruebas funcionales o pruebas no funcionales. En este caso, este capítulo se centra en las pruebas de sistema no funcionales y, en concreto, en las pruebas de rendimiento, prestando especial atención a los servicios web.

Al contrario que las pruebas funcionales, mostradas en los capítulos anteriores, en las que se busca encontrar divergencias entre los requisitos funcionales del sistema y el comportamiento observado, las pruebas no funcionales se encargan de comprobar características generales de un sistema software, como son la seguridad, la usabilidad, el rendimiento o la robustez, entre otros. Aunque las pruebas no funcionales son especialmente importantes en sistemas críticos, éstas no están limitadas a este tipo de sistemas.

Dentro de las pruebas no funcionales se encuentran las pruebas de rendimiento, cuyo objetivo es comprobar cómo se comporta el sistema bajo unas condiciones de carga determinadas, buscando defectos que sólo aparecen cuando el sistema tiene

que manejar dichas condiciones de carga durante un período de tiempo.

En este capítulo se describe una aproximación de pruebas basadas en propiedades que permitirá determinar el rendimiento de un sistema software desde una perspectiva de caja negra, en concreto, accediendo al mismo a través de un servicio web. Es importante destacar que, en este capítulo, únicamente se mostrarán las bases de esta aproximación y que aunque se ha utilizado un caso de estudio que ilustra cómo puede ser usada en sistemas software reales, es necesario evolucionar esta técnica para poder obtener más información acerca del rendimiento de un sistema software. Esta línea de investigación está actualmente en desarrollo por otros integrantes del grupo de investigación MADS.

El resto del capítulo se organiza de la siguiente manera. La sección 9.2 realiza una breve descripción de lo que son las pruebas de rendimiento, centrándose en el rendimiento de los servicios web. Posteriormente, la sección 9.3 describe la técnica propuesta, en la cual se usan propiedades para evaluar el rendimiento de servicios web, y se aplica a un caso de estudio real. Finalmente, la sección 9.4 resume los contenidos de este capítulo.

9.2. Las pruebas de rendimiento

El objetivo principal de las pruebas de rendimiento [268] es realizar una investigación técnica y empírica, a través de simulaciones basadas en la realidad, que permita ofrecer información acerca de la calidad de un producto software en términos del impacto causado por sus usuarios según diversas características, como son su velocidad, su escalabilidad o su estabilidad. Este tipo de pruebas, además de determinar cómo se comporta el sistema para una carga determinada ayudando a identificar cuellos de botella, también puede revelar errores de programación que típicamente aparecen sólo cuando el sistema tiene una carga alta, normalmente relacionados con el manejo de fallos, la gestión de recursos o algunas partes sensibles al manejo del tiempo. Por otro lado, los resultados obtenidos con las pruebas de rendimiento pueden ayudar a estimar la configuración de hardware necesaria para el despliegue del sistema software en el entorno de producción.

Las pruebas de rendimiento engloban a su vez otro tipo de pruebas más específicas, como son las pruebas de carga, las pruebas de resistencia o las pruebas de estrés. Así, las pruebas de carga tratan de mostrar el comportamiento del sistema bajo unas condiciones de carga realistas durante un período de tiempo, intentando encontrar cuál es la carga soportada por el sistema. Por otra parte, las pruebas de resistencia se realizan para comprobar si el sistema es capaz de soportar la carga esperada en el sistema de forma continuada durante un período largo de tiempo, con el objetivo de detectar defectos como la degradación del rendimiento o fugas de memoria. Por último, las pruebas de estrés son usadas para comprender cómo se comporta el sistema con unos niveles de carga mucho mayores a los esperados;

de esta forma, en lugar de tratar de encontrar la máxima carga sostenible, someten al sistema a una carga muy alta con el fin de evaluar su robustez en función de si éste se comporta de una manera aceptable. Entre los diferentes tipos de pruebas de rendimiento, este capítulo se centra fundamentalmente en las pruebas de carga.

Solucionar los problemas de rendimiento que aparecen en un sistema software suele ser un trabajo costoso, puesto que en algunas circunstancias se requiere el rediseño y reimplementación de algunas partes de dicho sistema, lo cual puede conllevar una serie de problemas asociados, como son retrasos en la planificación, sobrecostes o pérdida de productividad. Es por ello que existen enfoques, como la *ingeniería del rendimiento de software* (SPE) [325], que define un proceso sistemático y cuantitativo que comienza en las primeras etapas del ciclo de vida, estableciendo una serie de tareas a realizar a lo largo de las fases de requisitos, diseño, implementación, pruebas, despliegue y mantenimiento, para la construcción de sistemas software que cumplan con los objetivos de rendimiento.

De manera genérica, llevar a cabo las pruebas de rendimiento consiste en seguir el proceso compuesto por las siguientes actividades [262] (ver figura 9.1):

1. **Identificar el entorno de pruebas:** una de las primeras tareas a realizar es identificar cuál va a ser el entorno de pruebas y el entorno de producción (suponiendo que uno no va a ser una réplica del otro), teniendo en cuenta el hardware y software disponibles, así como las herramientas y recursos de prueba que se utilizarán. Conocer el entorno de pruebas desde las primeras etapas del proyecto ayuda a definir y diseñar los casos de prueba de una manera más eficiente, ayudando a identificar posibles dificultades que puedan surgir. En algunos casos, este proceso debe ser realizado de nuevo a lo largo del ciclo de vida del sistema software.

2. **Definir los criterios de aceptación:** se deben definir los objetivos de rendimiento del sistema software, como pueden ser tiempos de respuesta aceptables, utilización de recursos del sistema u otras restricciones relacionadas con el rendimiento.

3. **Diseñar los casos de prueba:** los casos de prueba deben ser diseñados para imitar el comportamiento del sistema software una vez desplegado en el entorno de producción, incluyendo algún tipo de aleatoriedad que simule la variabilidad en el comportamiento de los diferentes tipos de usuarios que pueden usar el sistema software. Además, se deben establecer qué métricas deben poder ser recopiladas una vez ejecutadas las pruebas.

   El diseño de los casos de prueba depende, en gran medida, de las herramientas específicas usadas para su ejecución (las cuales se establecen en el paso 1 de este proceso). Así, algunas herramientas [69, 151] se basan en modelar el comportamiento de los usuarios en el sistema definiendo la secuencia de operaciones (con una determinada variabilidad) que son invocadas por los

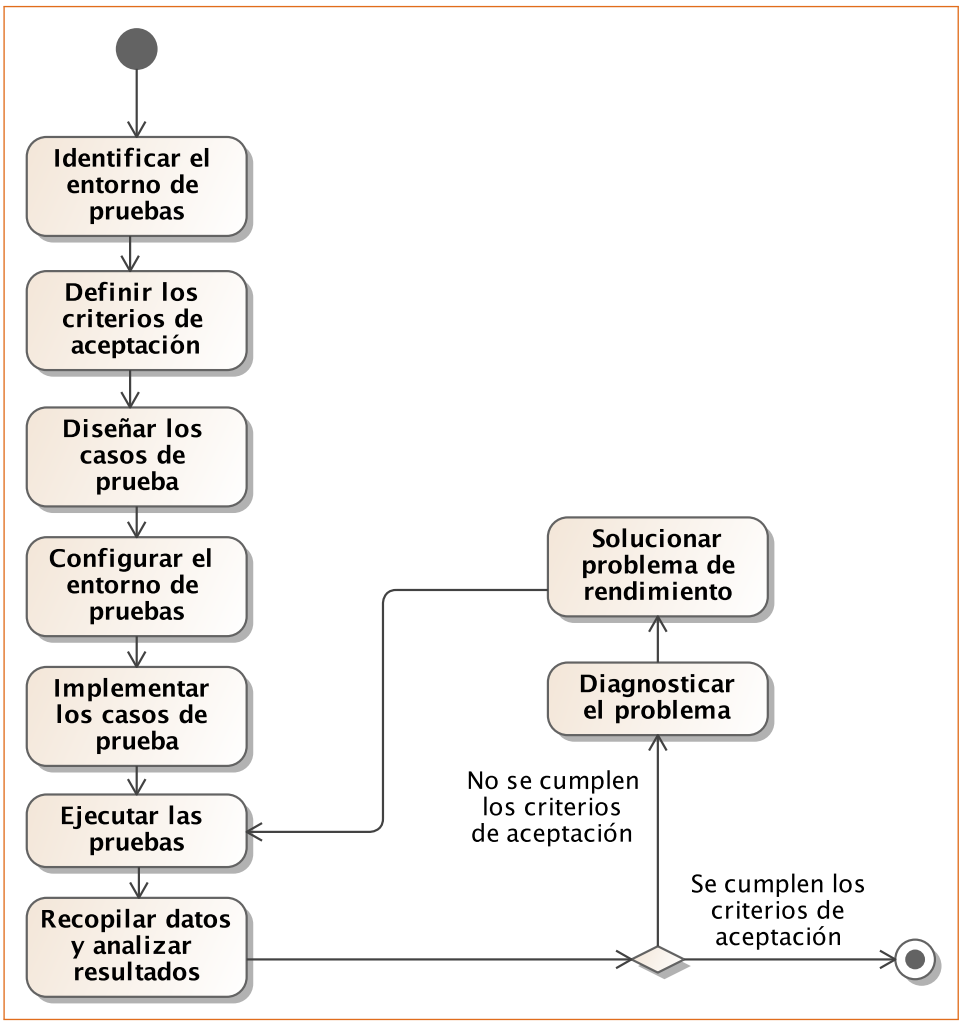


FIGURA 9.1: *Proceso para realizar las pruebas de rendimiento*

mismos, utilizando esta información para simular usuarios reales que usan el sistema. Otras aproximaciones se basan en la definición de tipos de eventos, indicando cuál es el número de veces que puede ocurrir cada uno de ellos en el sistema (por ejemplo, el número de veces por segundo que es invocada cada operación), y a partir de esta información estimular el sistema para aumentar su carga [283].

4. **Configurar el entorno de pruebas:** una vez diseñados los casos de prueba se debe preparar el entorno de pruebas, junto con las herramientas y recursos necesarios para ejecutar dichas pruebas, asegurando que éste está lo suficientemente monitorizado como para poder obtener los datos necesarios durante o tras su ejecución.

5. **Implementar los casos de prueba:** teniendo en cuenta el diseño de los casos de prueba y el entorno en el que se ejecutarán es posible implementar los

casos de prueba. Obviamente, esta tarea depende de la herramienta concreta que se usará para ejecutar las pruebas. En algunos casos, las pruebas de rendimiento se definen a través de una especificación escrita en XML [69], otras aproximaciones usan ficheros JSON [151] para dicha especificación y, en otros casos, se usan lenguajes específicos como, por ejemplo, TTCN-3 [319].

6. **Ejecutar las pruebas:** los casos de prueba se ejecutan usando las herramientas apropiadas para poder obtener los datos requeridos para su posterior análisis [312], como pueden ser Tsung [69], JMeter [8], LoadRunner [41], o la propia herramienta Megaload [151] (que se usará en este capítulo), entre otras.

7. **Recopilar datos y analizar resultados:** los resultados obtenidos de la realización de las pruebas deben ser analizados para comprobar si los valores obtenidos están dentro de los límites aceptables establecidos. Si después de recopilar y analizar toda la información deseada no se ha violado ninguno de los umbrales definidos, se pueden dar por concluidas las pruebas de rendimiento para un escenario particular en una configuración particular.

8. **Diagnosticar el problema:** si, por el contrario, en el paso anterior, alguno de los criterios definidos no se cumple, se debe diagnosticar el problema, el cual podría estar relacionado con un defecto en la aplicación, en el acceso a base de datos, en el uso de la red, o en cualquier otro aspecto del sistema. Para ello, existen herramientas que se pueden usar para realizar esta tarea, conocidas como *profilers* [201, 275], las cuales pueden a ayudar a realizar este diagnóstico.

9. **Solucionar el problema:** una vez detectado el problema, éste debe ser corregido, para lo cual podría ser necesario ajustar la configuración del sistema, corregir la implementación del sistema software o incluso refactorizar parte de dicho sistema software. Una vez realizada esta tarea, es posible ejecutar las pruebas de nuevo.

Entre los tipos de aplicaciones existentes en el mundo del software, las aplicaciones web y, en concreto, los servicios web, son uno de los objetivos candidatos a la realización de este tipo de pruebas [337]. En los sistemas web escalables es importante que, incluso cuando la popularidad del servicio aumenta y el número de usuarios se incrementa, los requisitos de rendimiento se sigan cumpliendo. Aunque el esquema de tareas básico mostrado en la figura 9.1 es válido para probar servicios web, existen numerosos trabajos relacionados con la realización de pruebas de rendimiento que tratan con las peculiaridades de este tipo de sistemas [98, 168, 263].

Las pruebas de rendimiento ofrecen la posibilidad de predecir cómo se comporta el sistema software con respecto a una serie de requisitos no funcionales, abordando diferentes aspectos, como son la evaluación de la capacidad del sistema para procesar peticiones simultáneas que proceden de cientos o miles de clientes, o cuál es la frecuencia de peticiones que el sistema puede manejar. Por tanto, medidas como

el número de peticiones por segundo que puede atender el servidor, la latencia de las peticiones tanto desde el punto de vista del servidor (tiempo entre que llega una petición hasta que es atendida) como del cliente (tiempo desde que el cliente realiza una petición hasta que recibe la respuesta), o la tasa de errores (peticiones rechazadas o no atendidas a tiempo), son parámetros que se deben tener en cuenta a la hora de realizar pruebas de rendimiento de servicios web. Además, cabe destacar que alguna de las medidas comentadas anteriormente, en especial la latencia percibida por el cliente, puede sufrir de efectos ajenos al sistema, como la saturación del canal de comunicaciones, por razones no achacables al sistema en sí.

Los servicios web son, de hecho, el tipo de sistema donde la aproximación de pruebas descrita en este capítulo será aplicada. Como se comentará en las siguientes secciones, se usará una aproximación basada en propiedades para automatizar lo máximo posible la ejecución de este tipo de pruebas.

9.3. Las pruebas de carga de servicios web basadas en propiedades

Las pruebas de carga se usan para determinar cómo se comporta un sistema ante diferentes niveles de uso. Sin embargo, uno de los inconvenientes de este tipo de pruebas es la gran cantidad de tiempo y esfuerzo necesario para llevarlas a cabo, por las siguientes razones:

- Por un lado, es necesario diseñar los casos de prueba, para lo cual se debe modelar cómo representar la carga del sistema, por ejemplo, describiendo el comportamiento de los usuarios reales que usen dicho sistema a probar. Esta tarea suele ser bastante costosa, puesto que se debe analizar qué tipos de usuarios utilizarán el sistema en producción así como modelar su comportamiento. Además, también se debe configurar el entorno de pruebas, que debe ser similar al entorno de producción, en el que se ejecutarán los casos de prueba implementados.
- Por otro lado, con respecto a la ejecución de las pruebas, para poder comprobar los diferentes aspectos relacionados con el rendimiento del sistema a probar, suele ser necesario llevar dicho sistema a situaciones límite, lo cual puede requerir también una gran cantidad de tiempo y, en ocasiones, también de recursos. Además, en muchas ocasiones, es necesario ejecutar las pruebas varias veces, con diferentes parámetros de configuración, para poder evaluar el rendimiento del sistema con diferentes configuraciones. Por tanto, en general, el tiempo requerido para llevar a cabo las pruebas de carga suele ser bastante grande.

Si bien es cierto que una vez diseñados e implementados los casos de prueba, la ejecución de las pruebas de carga en un entorno de pruebas configurado podría ser una tarea no asistida que puede prolongarse varios minutos, horas o incluso días,

en la práctica suele convertirse en una tarea que requiere interacción humana casi constante. Esto es debido a que las aproximaciones típicas para llevar a cabo las pruebas de carga consisten en incrementar progresivamente la carga del sistema a probar y observar el efecto producido. Por tanto, una vez obtenidos los resultados de las pruebas para una determinada configuración, se deben revisar y analizar para tomar una decisión acerca de si es necesario repetir las pruebas con una configuración diferente.

Por ejemplo, cuando se trata de sistemas web, la carga del sistema suele aumentar en función del número de usuarios que estén usando dicho sistema, por lo que, al probar este tipo de sistemas, la carga generada en estos casos dependerá el número y tipo de los usuarios simulados. Así, uno de los objetivos habituales de las pruebas de carga es averiguar cuál es el número máximo de usuarios concurrentes que puede manejar un sistema sin que exista una degradación significativa del comportamiento no funcional del mismo (velocidad, estabilidad, escalabilidad, etc.). En este ejemplo, el número y tipo de usuarios serán parámetros que se varíen manualmente entre varias ejecuciones de las pruebas.

Desde este punto de vista, es posible cambiar el enfoque manual descrito anteriormente por una aproximación basada en propiedades [252]. De esta forma, las propiedades definen el criterio de aceptación para dichas pruebas, y se ejecutarán con diferente número de usuarios, generados automáticamente a partir de un generador de datos. Así, se evita la necesidad de observar los resultados obtenidos manualmente para cada configuración de las pruebas, puesto que son los propios criterios de aceptación definidos en las propiedades, y los generadores de datos, los que guían la ejecución automática de las pruebas de carga con diferentes configuraciones. Para poder ejecutar las pruebas no funcionales siguiendo esta aproximación es necesario disponer de una especificación de pruebas que soporte la parametrización de algunos de sus aspectos, como pueden ser, dependiendo de la propiedad concreta a comprobar, la configuración del número máximo de usuarios concurrentes, la tasa de llegada de usuarios o el porcentaje de usuarios de un determinado tipo, entre otros.

Por tanto, comprobar cuál es este número máximo de usuarios concurrentes con una aproximación manual implica ejecutar las pruebas con diferentes configuraciones de la especificación, y observar manualmente los resultados obtenidos para poder llegar a una conclusión acerca de cuál es el máximo número de usuarios concurrentes soportados por el sistema. En cambio, con una aproximación basada en propiedades, se debe escribir una propiedad que exprese cómo se deben ejecutar las pruebas y qué comprobaciones hay que realizar para cada ejecución de las mismas, las cuales definen los criterios de aceptación. De esta forma, la propia herramienta de pruebas se encargará de ejecutar la propiedad y buscar el valor deseado, en este caso, el número máximo de usuarios concurrentes soportados. En concreto, la propiedad a escribir para este ejemplo sería la siguiente:

$$\forall n \in \mathbb{N}, \quad R = f(n) \Rightarrow post(R)$$

donde:

- n representa el número de usuarios concurrentes.
- f representa la ejecución de las pruebas de rendimiento con el número de usuarios concurrentes n.
- R es el resultado de la ejecución de dichas pruebas incluyendo datos como, por ejemplo, el tiempo medio de respuesta, el número de peticiones atendidas o el número de peticiones fallidas, entre otros.
- $post$ representa las comprobaciones a realizar sobre el resultado obtenido R, comparando dichos valores con los esperados, definiendo, de esta forma, un criterio de aceptación de las pruebas. Un ejemplo de comprobación podría ser:

  $$AvgRespTime < 3000 \wedge ConnRefused = 0 \wedge Timeouts = 0$$

  es decir, que el tiempo de respuesta medio sea inferior 3 segundos, y que no haya peticiones rechazadas ni peticiones no atendidas a tiempo.

Esta propiedad comprueba, por tanto, que para cualquier número de usuarios concurrentes, el sistema se comporta de la manera esperada, es decir, cumpliendo el criterio de aceptación definido con dicha propiedad, cuando está siendo usado por un número concreto de usuarios concurrentes.

Al contrario que en el uso de las pruebas basadas en propiedades para probar requisitos funcionales, donde se espera que la propiedad definida se cumpla para todos los casos de prueba generados; en este caso es lógico pensar que las comprobaciones a realizar para comprobar el comportamiento esperado del sistema fallen en algún momento, en concreto, cuando el número de usuarios concurrentes sea un valor más grande que el número máximo de usuarios concurrentes para los que el sistema software ha sido diseñado. Por tanto, en este caso, cuando se encuentre un valor para el que la propiedad no se cumple, se debe informar acerca de este valor. Sin embargo, esto no es suficiente, puesto que si, por ejemplo, el límite del sistema se encuentra entorno a los 1000 usuarios concurrentes, informar que el sistema no funciona de la manera adecuada para 8000 usuarios concurrentes no sería una información útil. Por tanto, es necesario proporcionar más información acerca de los límites del sistema, en este caso, el mínimo número de usuarios concurrentes para los que la propiedad no se cumple.

Por otro lado, se debe tener en cuenta que éste es un proceso que normalmente exhibe un comportamiento no determinista, es decir, el valor obtenido puede variar ligeramente entre varias ejecuciones de la propiedad. Factores externos no

controlados como la latencia variable en la realización de peticiones y obtención de respuestas del servidor debido al estado de la red, o la ejecución de otros procesos simultáneos en la misma máquina donde se encuentra instalado el sistema a probar, que pueden causar variaciones en el uso del procesador, la red o la memoria, son algunas de las causas por las que el resultado de estas pruebas puede variar entre ejecuciones. Sin embargo, esto no es un problema, puesto que en este tipo de pruebas no se busca un valor concreto para el que el sistema no cumple la propiedad. Así, cuando se quiere obtener información acerca de cuántos usuarios concurrentes son soportados por el sistema, no es necesario conocer un número exacto de usuarios, sino que se espera obtener un rango de usuarios para los cuales el sistema se comporta de una manera adecuada y, a partir del cual, el comportamiento del sistema empieza a no ser el adecuado.

Cuando se implementa esta aproximación usando herramientas concretas, se debe tener en cuenta que, aunque este tipo de pruebas basan su especificación en propiedades que describen el comportamiento a probar, no se comprueban que estas propiedades se cumplen para todos y cada uno de los casos de prueba posibles, sino que se seleccionan algunos casos de prueba y se ejecutan las propiedades únicamente para éstos. Además, se debe tener en cuenta que comprobar si la propiedad se cumple para una configuración concreta es un proceso lento, que puede durar varios minutos o incluso horas. Por ello, la generación de estos valores, en este ejemplo, el número de usuarios concurrentes, es un aspecto muy importante para que se realice una búsqueda eficiente del rango de valores para los cuales la propiedad especificada deja de cumplirse.

Teniendo en cuenta el proceso de realización de pruebas de rendimiento que se muestra en la figura 9.1, aunque con esta aproximación no se ahorra tiempo en las etapas iniciales de diseño e implementación de los casos de prueba, o la configuración del entorno de pruebas, el tiempo supervisado dedicado a la ejecución de las pruebas es mínimo, puesto que la propia herramienta de pruebas ejecutará las pruebas automáticamente con diferentes configuraciones con el objetivo de buscar la información deseada. De esta forma, se evita el trabajo manual de cambiar la configuración de las pruebas, ejecutarlas de nuevo, y analizar los resultados obtenidos para decidir si se deben ejecutar las pruebas con otra configuración.

A continuación se muestra cómo aplicar esta técnica con la herramienta Megaload, la cual se integra con QuickCheck, para llevar a cabo, usando una aproximación basada en propiedades, las pruebas de carga de servicios web [188].

9.3.1. Megaload y QuickCheck

Megaload [151] es una herramienta que permite realizar pruebas de rendimiento de servicios web, y ha sido construida durante la realización del proyecto europeo PROWESS. Las características más destacadas de esta herramienta son su escalabilidad y extensibilidad, así como el soporte para realizar pruebas de rendimiento en

la nube. Además, se integra con la herramienta QuickCheck, permitiendo el uso de una aproximación basada en propiedades para llevar a cabo pruebas de rendimiento.

La arquitectura de Megaload se basa en la definición de una serie de nodos, que conforman un clúster, los cuales simulan el comportamiento de los usuarios que usan el sistema. De esta forma, desde cada uno de estos nodos se invocarán operaciones del sistema a probar a través de un protocolo específico, y siguiendo un patrón de carga definido. Actualmente, los protocolos soportados son HTTP/1.1 y XMPP sobre BOSH y TCP, aunque es posible añadir soporte para más protocolos haciendo uso de la extensibilidad de dicha herramienta.

Por otra parte, Megaload es capaz de proporcionar información en tiempo real a medida que se ejecutan las pruebas, así como a la finalización de las mismas. Con respecto a la información que se puede obtener, Megaload ofrece una serie de métricas predefinidas a través de las cuales es posible obtener datos referentes a la ejecución de las pruebas, como el número de peticiones realizadas durante una prueba, el tiempo medio de respuesta o el porcentaje de peticiones correctamente atendidas, entre otras; pero también permite al usuario definir sus propias métricas. Estos datos pueden consultarse, por medio de gráficas e informes, a través de una interfaz de usuario que la propia herramienta proporciona para gestionar las pruebas.

La carga a generar por la herramienta Megaload en el sistema a probar, es decir, el comportamiento de los usuarios, se describe a través de una especificación de pruebas, escrita en el lenguaje JSON. En esta especificación se definen las operaciones del sistema que deben ser invocadas por Megaload. Para ello, es posible especificar una serie de fases secuenciales, cada una de ellas definiendo una tasa específica de llegada de usuarios, su duración y límites superiores para algunos parámetros, como son el número máximo de usuarios creados, el número máximo de usuarios concurrentes o la máxima tasa de peticiones por segundo realizadas durante dicha fase. Además, cada fase también tiene asociados un conjunto de escenarios. Cada escenario, por su parte, representa el comportamiento de un tipo de usuario que está usando el sistema, describiendo las operaciones que realizará un usuario de dicho tipo. Para la definición de este comportamiento es posible usar instrucciones de control de flujo, añadir cierta aleatoriedad en la invocación de las operaciones, definir períodos de inactividad variables durante los cuales el usuario no está realizando ninguna acción, llamar a funciones definidas en módulos externos o leer valores de un fichero CSV externo, entre otras acciones. Además, cuando se especifican los escenarios que forman parte de una fase, es posible especificar el porcentaje de aparición de cada uno de ellos. Estos porcentajes representan, por tanto, cuántos usuarios de cada tipo usarán el sistema a probar.

Además de la escalabilidad que ofrece la herramienta Megaload a través de la definición de múltiples nodos cliente que permiten estimular al sistema a probar, y su extensibilidad en cuanto a la definición del soporte de nuevos protocolos y

métricas, destaca el soporte que proporciona la propia herramienta Megaload para realizar las pruebas de rendimiento usando una aproximación basada en propiedades. En concreto, Megaload se integra con la herramienta QuickCheck. Por tanto, es posible formular propiedades que usan la información ofrecida por las diferentes métricas que proporciona Megaload para decidir si el sistema satisface una serie de requisitos no funcionales. Por ejemplo, la implementación de la propiedad descrita anteriormente para obtener el número máximo de usuarios concurrentes soportados por el sistema podría ser escrita en QuickCheck de la siguiente manera:

```
prop_max_concurrent_users() ->
 ?FORALL(MaxProc, eqc_gen:nat(),
  begin
   loader:stop_load(),
   megautils:wait_for_sut_stabilization(),
   megautils:load_configuration(MaxProc),
   case loader:start_load(?TEST_SPEC) of
    {ok,ok} ->
     megautils:wait_until_terminated(),
     ResTime = lmh:getht_res_time(),
     AvgRespTime = proplists:get_value(
        arithmetic_mean, ResTime),
     ConnRefused = lmh:getc_conn_refused(),
     Timeouts = lmh:getc_timeout(),
     AvgRespTime < 3000000 andalso
        ConnRefused == 0 andalso Timeouts == 0;
    Other ->
     false
   end
  end).
```

Esta propiedad general, después de parar cualquier otra prueba en ejecución usando la función `stop_load` y esperar a que el sistema a probar se estabilice (`wait_for_sut_stabilization`), configura la especificación de pruebas, usando para ello la función `load_configuration`, con el número de usuarios concurrentes correspondiente (generado automáticamente con el generador `nat` proporcionado por QuickCheck que produce enteros positivos mayores que cero). Posteriormente, se comienza la ejecución de las pruebas, para lo que se usa la función `start_load` proporcionada por Megaload, que recibe como parámetro el fichero principal donde se describe, usando el lenguaje JSON, la especificación de las pruebas a realizar, y se espera a que termine su ejecución. Este proceso puede durar varios minutos o incluso horas. Una vez que terminada la ejecución de las pruebas, se obtienen los valores deseados usando el módulo `lmh` proporcionado por Megaload, y se realizan las comprobaciones oportunas, en este caso, que el tiempo medio de respuesta es menor a 3 segundos, que no se han rechazado conexiones, y que no

se han producido errores por peticiones no atendidas a tiempo. Si alguna de estas condiciones no se cumple, la propiedad fallará, y QuickCheck comenzará el proceso de reducción para obtener el mínimo caso para el que esta propiedad no se cumple, lo cual se corresponde con el número máximo de usuarios concurrentes soportados.

Como ya se ha comentado anteriormente, el uso de una aproximación basada en propiedades para la realización de pruebas no funcionales varía ligeramente de cómo se usa para la realización de pruebas funcionales. Así, mientras que en las pruebas funcionales el resultado esperado es que las propiedades definidas se cumplan para todos los casos de prueba generados, en las pruebas de carga se espera que las propiedades definidas no se cumplan para una determinada configuración, a partir de la cual se debe buscar cuáles son los límites del sistema a probar. Por tanto, en este caso, no sería útil que QuickCheck informase que el sistema no funciona como se espera para un número muy elevado de usuarios concurrentes, para el cual ya se esperaba un comportamiento erróneo, sino que debería informar sobre los límites del mismo. Además, teniendo en cuenta que la ejecución de este tipo de pruebas es una tarea lenta, es muy importante que esa búsqueda de la configuración límite sea óptima.

Por este motivo, la implementación de la propiedad definida anteriormente, es decir, `prop_max_concurrent_users`, no es lo más adecuado para obtener la información deseada. La tabla 9.1 muestra un ejemplo de distribución obtenida para una generación de 100 casos de prueba con el generador `nat` usado en esta propiedad. Si se analizan estos valores, se puede observar que todos ellos son valores muy bajos, no muy útiles, en general, cuando se intenta averiguar el número máximo de usuarios concurrentes soportados por un sistema software. Evidentemente, podría multiplicarse el valor generado por alguna cantidad, pero aún así la búsqueda del número máximo de usuarios concurrentes soportados no sería muy eficiente. También se podría usar algún otro generador proporcionado por QuickCheck, como `choose`, el cual genera valores en un intervalo determinado. Sin embargo, en este último caso, se debe conocer cuál es un buen intervalo de valores posibles para usar en las pruebas. Además, usando este generador, los datos se generan usando una distribución aleatoria uniforme, donde cada valor del rango especificado tiene la misma probabilidad de ser elegido, sin seguir ningún orden que guíe la búsqueda de los límites del sistema.

Por esta razón, como la aleatoriedad de los generadores proporcionados por defecto por QuickCheck no es adecuada en este caso, ha sido necesario usar generadores de datos propios [86]. Así, la idea en este tipo de pruebas es comenzar con una carga baja, e ir aumentándola progresivamente a través de diferentes estrategias de crecimiento, por ejemplo, lineal o exponencial, hasta alcanzar un valor para el cual la propiedad no se cumple. Una vez alcanzado este punto, se debe realizar el proceso contrario, es decir, reducir progresivamente la carga hasta encontrar el límite del sistema. Este proceso de reducción es el mismo que se usa para mostrar un caso de

Valor	Porcentaje
5	10 %
4	10 %
0	10 %
3	9 %
1	9 %
6	7 %
8	6 %
17	4 %
10	4 %

Valor	Porcentaje
9	4 %
22	3 %
21	3 %
18	3 %
2	3 %
23	2 %
20	2 %
16	2 %
7	2 %

Valor	Porcentaje
26	1 %
19	1 %
15	1 %
14	1 %
13	1 %
12	1 %
11	1 %

TABLA 9.1: *Ejemplo de distribución de 100 valores producidos por el generador* `nat` *de QuickCheck*

prueba más "pequeño" que provoca que una propiedad no se cumpla a partir de un caso de prueba generado para el cual la propiedad ha fallado previamente. En este caso también se ha redefinido este proceso para los generadores empleados, el cual se ha implementado como una búsqueda binaria, donde el siguiente valor usado en las pruebas se calcula según la siguiente fórmula:

$$Min + ((Max - Min)/2)$$

donde Min y Max representan los dos últimos valores comprobados para los que la prueba se cumple y no se cumple respectivamente, o viceversa.

Por ejemplo, la tabla 9.2 muestra la secuencia de valores generados con esta aproximación si se usa una estrategia de generación exponencial de base 10. En este ejemplo, se ha supuesto que la propiedad especificada no se cumple cuando el número de usuarios concurrentes es mayor o igual que 2702. Sin embargo, en casos reales, la propiedad no siempre fallará con el mismo número de usuarios, es decir, existe un no determinismo asociado en la ejecución de las pruebas de rendimiento. Este hecho provoca que no sea interesante devolver como resultado de ejecutar las propiedad un valor concreto, como ocurría en las pruebas funcionales, sino que se espera que el resultado de estas pruebas sea un rango de valores. Por tanto, en los generadores definidos se permite usar un margen de tolerancia que se usará para construir dicho rango de valores. Por ejemplo, no tiene sentido informar que un sistema soporta 2702 usuarios concurrentes, puesto que este valor puede variar de una ejecución a otra, sino que una respuesta válida al número máximo de usuarios concurrentes soportados por el sistema podría ser un intervalo entre 2500 y 3000 usuarios concurrentes. Además, teniendo en cuenta este margen se ahorran pasos en la ejecución de las pruebas, requiriendo menos tiempo en la ejecución de las

mismas. La tabla 9.3 muestra de nuevo el mismo ejemplo de generación, pero asumiendo que se usa un margen de 100 usuarios. En este caso, el resultado obtenido sería el intervalo (2687, 2757), realizando 6 pasos menos.

Paso	Valor	Resultado	Etapa
1	1	`true`	Generación
2	10	`true`	Generación
3	100	`true`	Generación
4	1000	`true`	Generación
5	10000	`false`	Generación
6	5500	`false`	Reducción
7	3250	`false`	Reducción
8	2125	`true`	Reducción
9	2687	`true`	Reducción
10	2968	`false`	Reducción
11	2827	`false`	Reducción
12	2757	`false`	Reducción
13	2722	`false`	Reducción
14	2704	`false`	Reducción
15	2695	`true`	Reducción
16	2699	`true`	Reducción
17	2701	`true`	Reducción
18	2702	`false`	Reducción

TABLA 9.2: *Ejemplo de generación de valores con un generador exponencial de base 10 para realizar las pruebas de rendimiento de un sistema con un límite de 2702*

Por otro lado, dependiendo del sistema específico a probar, cabe destacar que, en algunos casos, es más óptimo usar unos generadores de datos u otros. Así, un generador exponencial de base 2 o un generador lineal con un incremento determinado también son estrategias perfectamente válidas para este tipo de pruebas. De la misma forma, el uso de márgenes más o menos grandes dependerá de la precisión con la que se desee obtener el resultado de las pruebas.

9.3.2. Caso de estudio: API de acceso a dispositivos proporcionada por el componente VoDKATV-core

Como se muestra en la figura 3.6 de la página 56, los dispositivos que acceden al sistema VoDKATV lo hacen usando una API de integración que proporciona el componente VoDKATV-core. Dicha API de integración es un servicio web que maneja y devuelve datos formateados en el lenguaje JSON. Al contrario que otros

Paso	Valor	Resultado	Etapa
1	1	`true`	Generación
2	10	`true`	Generación
3	100	`true`	Generación
4	1000	`true`	Generación
5	10000	`false`	Generación
6	5500	`false`	Reducción
7	3250	`false`	Reducción
8	2125	`true`	Reducción
9	2687	`true`	Reducción
10	2968	`false`	Reducción
11	2827	`false`	Reducción
12	2757	`false`	Reducción

TABLA 9.3: *Ejemplo de generación de valores con un generador exponencial de base 10 y un margen de tolerancia de 100 para realizar las pruebas de rendimiento de un sistema con un límite de 2702*

servicios web proporcionados por el sistema VoDKATV, como son los accedidos por aplicaciones de administración, en este caso sí es importante que este servicio soporte un gran número de usuarios usándolo simultáneamente. Por lo tanto, probar que este servicio web puede soportar un número determinado de usuarios concurrentes, sin que el funcionamiento de dicho servicio se vea alterado gravemente, es un aspecto importante dentro del sistema VoDKATV.

Cabe destacar que el resultado de estas pruebas dependerá, evidentemente, de la capacidad de las máquinas físicas en las que el sistema VoDKATV se encuentre instalado, así como de la distribución de los componentes de la arquitectura de VoDKATV en dichas máquinas. Desde este punto de vista, las configuraciones más habituales del sistema VoDKATV en despliegues reales son las siguientes:

- La configuración básica usa un contenedor de aplicaciones J2EE, en concreto, Apache Tomcat [9], y una base de datos relacional para almacenar la información, normalmente PostgreSQL [55]. En este caso, los dispositivos se conectan directamente al contenedor de aplicaciones J2EE.

- El sistema VoDKATV soporta el uso de un sistema de caché externo, en concreto, Memcached [43], el cual, como se mostrará más adelante, permite incrementar el rendimiento del sistema.

- También es muy común usar un servidor web, como, por ejemplo, nginx [47], que actúe como *proxy* entre los dispositivos y el contenedor de aplicaciones. Además, este sistema también se usa por razones de seguridad, tales como

restringir múltiples accesos desde la misma dirección IP, o para mejorar el rendimiento, por ejemplo, usando una caché para almacenar temporalmente el resultado de la ejecución de algunas operaciones costosas.

- Finalmente, es posible usar en el mismo despliegue VoDKATV varios contenedores de aplicaciones balanceados por un servidor web, como es nginx. Este es el tipo de despliegue más complejo usado hasta ahora en las instalaciones de VoDKATV, en concreto, en aquellas en las que se necesita soporte para un gran número de usuarios concurrentes. La figura 9.2 muestra el diagrama de despliegue que representa esta situación.

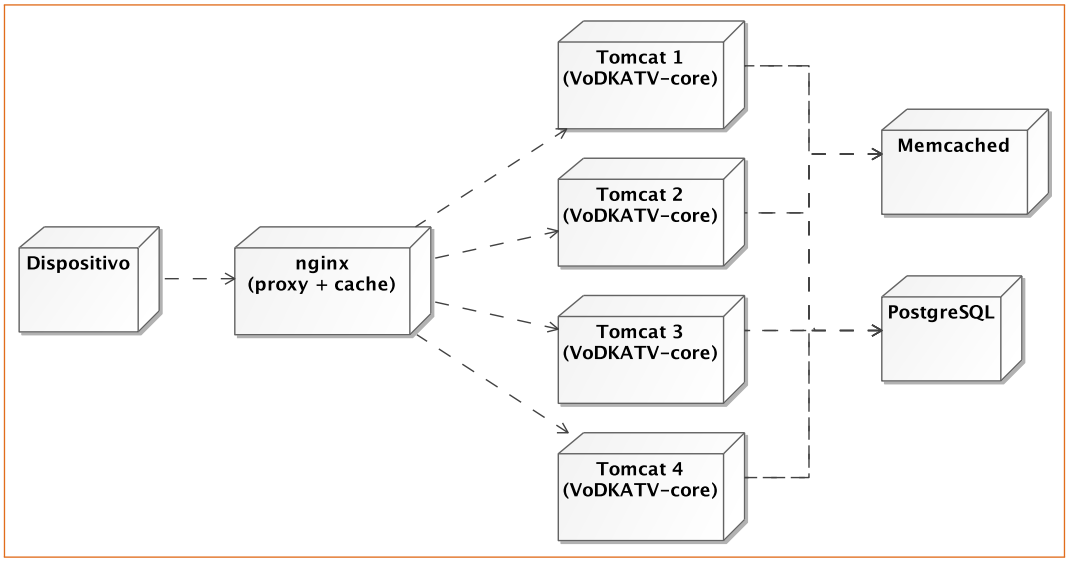


FIGURA 9.2: *Diagrama de despliegue en clúster del sistema VoDKATV*

En función del despliegue concreto, sobre todo teniendo en cuenta el número de usuarios que utilizarán el sistema, se usará una configuración u otra. En este caso, las propiedades implementadas se han ejecutado con la primera y segunda opción, puesto que será suficiente para ilustrar cómo utilizar la aproximación propuesta, basada en propiedades, para realizar las pruebas de rendimiento.

Con respecto al servicio web, cabe mencionar que el acceso al mismo requiere autenticación previa. Así, el flujo habitual de uso es invocar la operación *login* para iniciar una sesión en el sistema. Esta operación devuelve una *cookie* y un *token*, que deberán ser utilizados para invocar el resto de operaciones del servicio web, bien usando dicha *cookie* de la manera habitual dentro del protocolo HTTP, o bien usando el *token* como un parámetro a la hora de invocar el resto de operaciones. Por su parte, las operaciones invocadas después de iniciar una sesión en el sistema se usan para recuperar información que, en muchas ocasiones, se mostrará al usuario en el dispositivo correspondiente. De esta forma, el sistema VoDKATV proporciona

operaciones para obtener la lista de canales de televisión, la información sobre la programación de cada canal, el listado de contenidos bajo demanda, alquilar contenidos, obtener la información del usuario, cambiar las preferencias del usuario, etc. Finalmente, se debería invocar la operación *logout*, la cual cierra la sesión del usuario en el sistema, eliminando la sesión web y liberando recursos en el servidor. Como es habitual en aplicaciones web, si esta operación no se ejecuta, estos recursos serán liberados automáticamente después de un tiempo de inactividad.

También cabe destacar que en un despliegue típico del sistema VoDKATV suele haber tres perfiles diferentes de usuarios que usan la plataforma:

- Usuarios que inician una sesión en el sistema, pero abandonan su sesión después de unos pocos segundos.
- Usuarios que inician una sesión en el sistema, seleccionan un canal de televisión, y visualizan dicho canal durante un período largo de tiempo (minutos o incluso horas), después del cual seleccionan otro canal, y así sucesivamente.
- Usuarios que inician una sesión en el sistema e interactúan continuamente con el mismo. Por ejemplo, estos usuarios suelen cambiar continuamente de canal, consultar de manera habitual la programación de los canales, consultar de manera periódica si existen nuevos contenidos multimedia para alquilar y, en general, tienen una gran interacción con la interfaz del sistema.

El porcentaje de estos tipos de usuarios suele ser conocido de antemano antes de desplegar un sistema VoDKATV. Aunque estos porcentajes varían en función de la instalación concreta, en general, el número de usuarios del tercer tipo suele ser menor que el de los usuarios del primer tipo, y estos, a su vez, suelen ser menos que los usuarios del segundo tipo.

Cada uno de los tipos de usuarios descritos anteriormente tienen asociados diferentes patrones de comportamiento, es decir, cada uno de ellos provoca que se invoquen diferentes operaciones del servicio web, y con diferente frecuencia. De esta forma, cada tipo de usuario generará, potencialmente, diferentes niveles de carga en el servidor. Por tanto, una vez conocidos los tipos de usuarios que usarán el sistema, es importante analizar su comportamiento. Así, para el caso del sistema VoDKATV, se ha analizado el tráfico entre una aplicación cliente, en concreto, una aplicación web visualizada desde un navegador web de un ordenador, y el servidor VoDKATV-core. Para ello, se ha simulado manualmente el comportamiento de cada uno de los tipos de usuarios descritos anteriormente, y se han obtenido las URLs invocadas durante este proceso, así como los tiempos de espera entre estas invocaciones.

Con esta información a modo de muestra es posible escribir la especificación de pruebas, en formato JSON, para que pueda ser usada por Megaload. En esta especificación se deben definir una serie de escenarios, en concreto, uno por cada

uno de los perfiles o tipos de usuarios descritos anteriormente. Por tanto, en este caso, se han definido tres escenarios diferentes:

- `short-lived_user`: que representa los usuarios que abandonan la aplicación en unos pocos segundos.
- `tv_user`: representa los usuarios que simplemente ven la televisión, con muy pocas interacciones con la interfaz de usuario.
- `epg_user`: representa los usuarios que realizan múltiples interacciones con el sistema, sobre todo consultando la guía de programas.

Así, basándose en el análisis de comportamiento de cada uno de los tipos de usuarios, en cada uno de estos escenarios se especifica la secuencia correspondiente de URLs a invocar, así como tiempos de espera entre la invocación de cada una de ellas, simulando tanto las interacciones del usuario con el sistema como el tiempo que el usuario no está realizando ninguna acción.

Cuando se usa una aproximación manual, es común incluir varias fases en la especificación de pruebas, de tal forma que cada una de las fases tiene asociada una configuración determinada, como es el número máximo de usuarios creados, la tasa de llegada de usuarios o el porcentaje de cada tipo de usuarios. Así, una vez ejecutadas las pruebas se analizan los resultados devueltos, en forma de gráficas, informes o *logs*, para llegar a una conclusión acerca del rendimiento del mismo. En este caso, al usar una aproximación basada en propiedades, donde la especificación de pruebas se ejecutará múltiples veces con diferentes configuraciones generadas y cargadas automáticamente, se ha optado por crear una única fase cuya configuración se generará automáticamente en función de los generadores de datos usados en las propiedades especificadas, las cuales se describen a continuación.

9.3.2.1. Obtención del número máximo de usuarios concurrentes

Esta sección describe los resultados obtenidos en la implementación de la propiedad que permite obtener el número máximo de usuarios concurrentes soportados por el sistema VoDKATV [88]. Para ello, se tomará como criterio de aceptación que el tiempo de respuesta medio de las peticiones al servicio web sea menor que X milisegundos, es decir:

$$\forall n \in \mathbb{N}, \quad R = f(n) \Rightarrow AvgRespTime(R) < X$$

También podría ser interesante comprobar otros parámetros como conexiones rechazadas o conexiones no atendidas a tiempo, pero no se ha hecho puesto que, de esta forma, será posible mostrar de forma gráfica la evolución del tiempo de respuesta con respecto al número de usuarios concurrentes. Además, se han realizado

pruebas teniendo en cuenta estos parámetros y los resultados, en el sistema VoDKATV, fueron iguales o muy similares a los obtenidos con esta propiedad en todos los casos. Esto es debido a que este tipo de errores aparecen sólo cuando el número de usuarios concurrentes es muy grande y el tiempo de respuesta se incrementa mucho, por lo que la condición ya no se cumpliría de todos modos.

Como se ha comentado anteriormente, uno de los puntos clave en este proceso es el uso de un generador de valores adecuado. En este caso, se ha usado un generador exponencial de base 10 y un margen de tolerancia de 100 usuarios en el proceso de reducción, el cual se ha implementado como una búsqueda binaria. Así, la aproximación seguida consiste en que para cada uno de los valores generados (1, 10, 100, 1000, etc.) se configura la especificación de Megaload para que utilice dicho valor como el número máximo de usuarios concurrentes, se ejecutan las pruebas con la configuración resultante, y comprueban si el tiempo medio de respuesta obtenido como resultado ($AvgRespTime$) es menor al tiempo máximo establecido.

Cabe mencionar que en esta especificación de pruebas se ha usado el siguiente porcentaje de escenarios, por ser una de las distribuciones más comunes en la práctica:

- `short-lived_user`: 30 %.
- `tv_user`: 50 %.
- `epg_user`: 20 %.

Para la ejecución de estas pruebas se ha adaptado la propiedad genérica descrita anteriormente para que ejecute la especificación de pruebas de VoDKATV. Cabe destacar que la función `wait_for_sut_stabilization` se ha implementado simplemente esperando 60 segundos, puesto que resultados empíricos han mostrado que este tiempo es suficiente para que el sistema VoDKATV finalice las conexiones de los usuarios que estaban usando el sistema. Sin embargo, otras posibles aproximaciones podrían ser comprobar que el tiempo de respuesta del sistema vuelve a ser el esperado para volver a comenzar una nueva prueba. Además, se han usado diferentes valores para el tiempo máximo de respuesta permitido, específicamente, 1 o 3 segundos, y para la duración de cada una de las ejecuciones (30 segundos o 600 segundos).

Cabe mencionar también que esta propiedad ha sido evaluada con dos de las configuraciones típicas del sistema VoDKATV: usando simplemente un contenedor de aplicaciones Apache Tomcat, y usando dicho contenedor de aplicaciones junto con un servidor Memcached.

Así, la siguiente tabla muestra los resultados obtenidos usando un generador exponencial de base 10 con un margen de tolerancia de 100 usuarios, cuando se usa simplemente un contenedor de aplicaciones Apache Tomcat, teniendo en cuenta los diferentes tiempos máximos de respuesta admitidos (1 o 3 segundos) y las diferentes

duraciones de las pruebas (30 segundos o 600 segundos):

Tiempo de respuesta	1 seg.	1 seg.	3 seg.	3 seg.
Duración	**30 seg.**	**600 seg.**	**30 seg.**	**600 seg.**
Usuarios concurrentes	(100, 156)	(381, 437)	(212, 268)	(606, 662)
Pasos generación	4	4	4	4
Pasos reducción	4	4	4	4
Total pasos	8	8	8	8

TABLA 9.4: *Número máximo de usuarios concurrentes del sistema VoDKATV usando únicamente un contenedor de aplicaciones Apache Tomcat*

Como es lógico, cuanto más grande es el tiempo de respuesta permitido, mayor es el número de usuarios concurrentes soportados. También se ha observado que una duración corta para las pruebas no suele proporcionar resultados precisos. La explicación para este último punto es que la gran mayoría de los usuarios se crean al inicio de la prueba, y realizan una serie de peticiones para autenticarse en el sistema, las cuales, como se comentará posteriormente, son bastante costosas en comparación con otras operaciones. Una vez autenticados, los usuarios seguirán realizando peticiones, pero de forma mucho más esporádica. De esta forma, cuando la duración de la prueba es muy corta, las sesiones de muchos de los usuarios creados no podrán ser ejecutadas completamente, y el tiempo de respuesta medio está muy condicionado por el tiempo de las peticiones iniciales, por lo que es necesario un tiempo más largo para obtener resultados más precisos.

Además, durante la ejecución de la propiedad, los tiempos medios de respuesta obtenidos para cada caso de prueba se registran con el fin de generar una gráfica que pueda ayuda a comprender el comportamiento del sistema. Por ejemplo, la figura 9.3 muestra los resultados obtenidos durante la ejecución de la propiedad con un límite de tiempo de respuesta de 1 segundo, y una duración de las pruebas de 30 segundos. En este caso, en esta gráfica se muestran los resultados de cinco ejecuciones, junto con la media obtenida del total de las mismas. Como se observa en la gráfica, al usar un generador de datos exponencial de base 10, el número de usuarios concurrentes con los que se ejecuta la propiedad son 1, 10, 100 y 1000. Para 1, 10 y 100, el tiempo medio de respuesta obtenido es menor que el límite establecido de 1 segundo, mostrado con una línea roja en la gráfica, mientras que con 1000 se obtiene un tiempo de respuesta mucho mayor, de unos 11 segundos, que excede el límite establecido. En este punto, QuickCheck comienza el proceso de reducción en el que se intenta averiguar, con una búsqueda binaria, el número de usuarios con el que la propiedad se deja de cumplir. En concreto, se prueba con los valores intermedios entre 100 y 1000 mostrados en la gráfica: 550, 325, 212 y 156. Como se puede observar, el tiempo de respuesta de las peticiones al sistema VoDKATV aumenta de una forma casi exponencial a medida que el número de usuarios se incrementa.

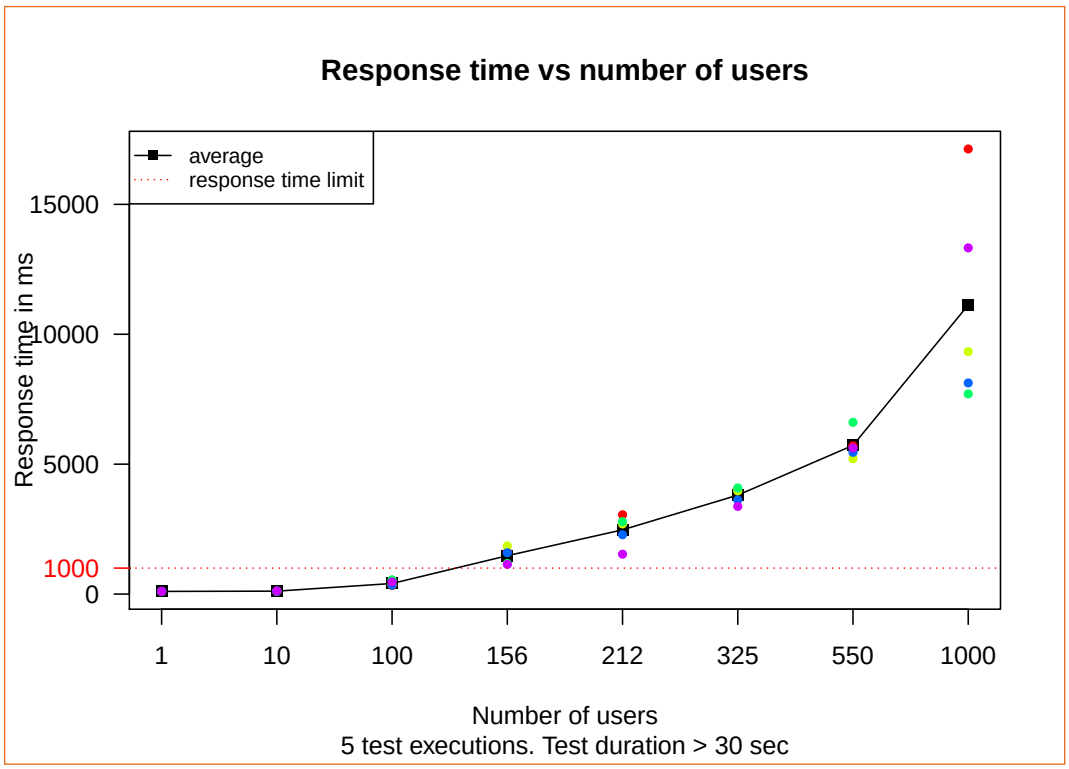


FIGURA 9.3: *Número máximo de usuarios concurrentes del sistema VoDKATV*

Con respecto al uso de un sistema de caché externo, en concreto, Memcached, los resultados obtenidos al volver a ejecutar la misma propiedad se muestran en la siguiente tabla:

Tiempo de respuesta	**1 seg.**	**1 seg.**	**3 seg.**	**3 seg.**
Duración	**30 seg.**	**600 seg.**	**30 seg.**	**600 seg.**
Usuarios concurrentes	(437, 493)	(887, 943)	(550, 606)	(1070, 1140)
Pasos generación	4	4	4	5
Pasos reducción	4	4	4	7
Total pasos	8	8	8	12

TABLA 9.5: *Número máximo de usuarios concurrentes del sistema VoDKATV usando un contenedor de aplicaciones Apache Tomcat y Memcached*

Como se observa, ahora el límite está en torno a los 1000 usuarios concurrentes, es decir, un 66 % más que con la configuración anterior. Por tanto, el sistema VoDKATV empieza a comportarse con lentitud con unos 400 usuarios más que en la configuración simple. Esto es debido a que el sistema Memcached permite almacenar cierta información en memoria, evitando múltiples accesos a base de datos y

otros sistemas externos, así como la realización reiterada de operaciones costosas.

Por otro lado, además de averiguar el rendimiento del servicio web en términos del número máximo de usuarios concurrentes, otra de las preguntas clave es conocer si esta degradación del servicio web a partir de un cierto número de usuarios afecta realmente a la interfaz de usuario que utiliza este servicio web, que es usada por los usuarios para acceder a la plataforma (visualizar canales, alquilar contenidos bajo demanda, cambiar las preferencias, etc.). Así, es importante conocer si a partir de 1000 usuarios concurrentes, los usuarios que usen la interfaz web van a tener una mala experiencia con la misma en cuanto a términos de lentitud.

Para responder a esta pregunta se hizo uso uso de una de las características ofrecidas por Megaload: se han etiquetado las URLs. Así, algunas URLs se han marcado como "importantes" y otras como "secundarias". Las URLs importantes son aquellas que son invocadas a partir de una acción que realiza el usuario y, por tanto, deben devolver resultados de forma inmediata. Por su parte, las URLs secundarias son aquellas invocadas por la interfaz de usuario de manera automática en segundo plano, y su lentitud no afecta de forma considerable al funcionamiento de la interfaz de usuario. Por ejemplo, las URLs que devuelven la lista de canales o la programación de cada canal pertenecen a este grupo, y son usadas por la interfaz web para actualizar esta información en segundo plano de forma periódica.

De esta forma, en los criterios de aceptación de las pruebas, únicamente se ha tenido en cuenta el tiempo medio de las URLs importantes. Con este criterio, los resultados obtenidos para un tiempo de respuesta límite de 3 segundos y una duración de las pruebas de 600 segundos, revelan que la interfaz de usuario podría estar siendo usada por unos 1300 usuarios concurrentes, es decir, aproximadamente un 30 % más de usuarios que los obtenidos anteriormente. Esta información puede ser utilizada para ajustar mejor la infraestructura concreta en un despliegue específico, así como comprometer un mejor servicio con unas ciertas garantías.

9.3.2.2. Obtención del número máximo de usuarios concurrentes por cada tipo de usuario

Otro ejemplo de propiedad que puede ser analizada es la obtención del número máximo de usuarios soportados por el sistema VoDKATV para cada uno de los tipos de usuarios descritos anteriormente [88]. Así, mientras que en la propiedad anterior se ha considerado una distribución típica de tipos de usuarios, en este caso será posible extraer conclusiones acerca de la carga generada por cada uno de estos tipos individualmente, las cuales son imposibles de obtener a partir del uso de escenarios mixtos. De esta forma, se ha repetido la ejecución de la propiedad anterior para cada uno de los tipos de usuarios considerados, es decir:

$$\forall e \in \{\texttt{tv\_user}, \texttt{epg\_user}, \texttt{short-lived\_user}\},$$
$$[\forall n \in \mathbb{N},\ \ R = f(e,n) \Rightarrow AvgRespTime(R) < X]$$

Así, cuando se ejecuta esta propiedad, la especificación de pruebas debe ser configurada para que use un número máximo de usuarios concurrentes y un único escenario: `tv_user`, `epg_user` o `short-lived_user`.

Estas pruebas se han ejecutado usando un contenedor de aplicaciones Apache Tomcat junto con un servidor Memcached. Al igual que las pruebas anteriores, se ha usado un generador exponencial para generar el número de usuarios concurrentes, con un algoritmo de búsqueda binaria en el proceso de reducción, y con un margen de tolerancia de 100 usuarios. Además, en este caso, únicamente se ha usado un límite de 3 segundos para el tiempo máximo de respuesta medio admitido, y una duración de cada prueba de 600 segundos. Las figuras 9.4, 9.5, y 9.6 muestran las gráficas resultantes de ejecutar esta propiedad con cada uno de los tres tipos de escenarios. Por otro lado, la figura 9.7 muestra el valor medio obtenido para cada escenario.

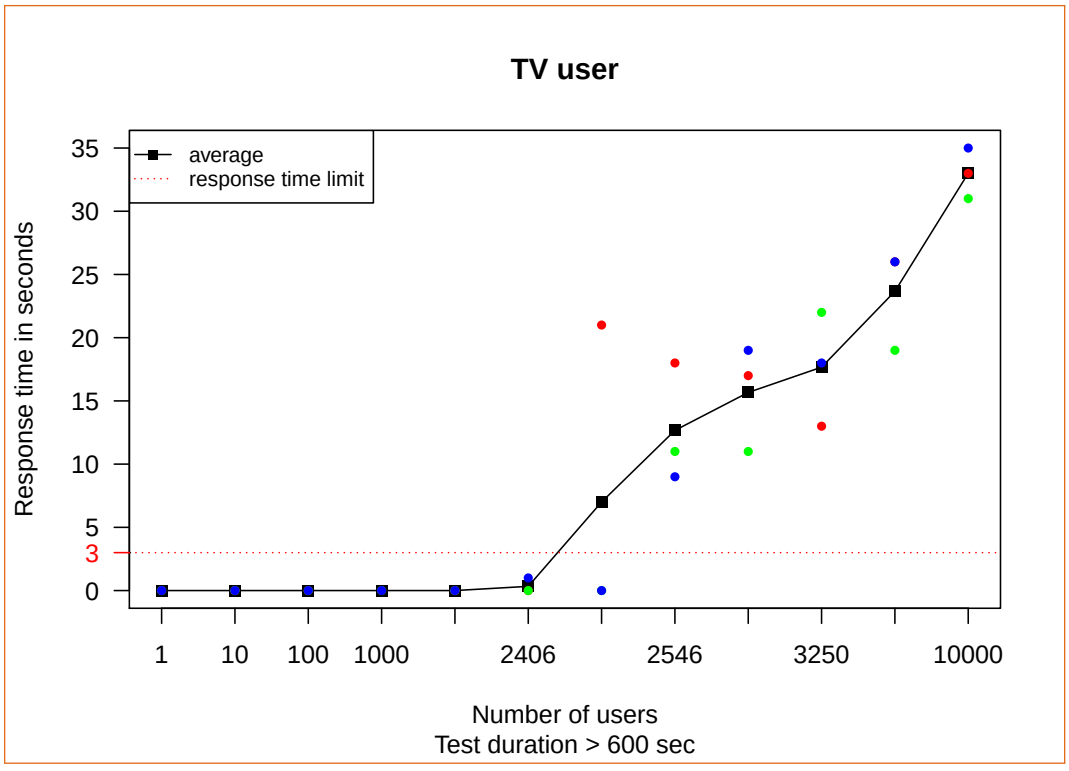


FIGURA 9.4: *Número máximo de usuarios concurrentes del sistema VoDKATV de tipo* `tv_user`

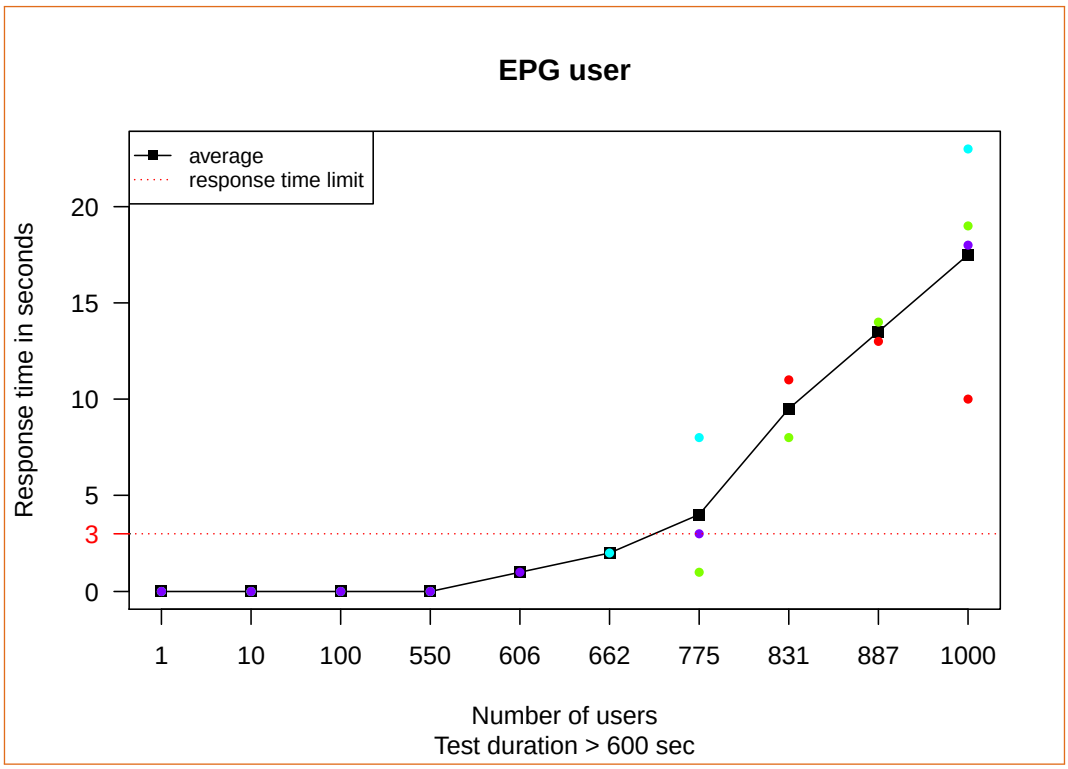


FIGURA 9.5: *Número máximo de usuarios concurrentes del sistema VoDKATV de tipo* `epg_user`

Teniendo en cuenta estas gráficas, se podría concluir que, aunque la evolución del tiempo del respuesta en función al tipo de usuario es similar en todos los casos, el sistema VoDKATV soporta más usuarios concurrentes del tipo `tv_user` que otros tipos de usuarios. Esto parece lógico, puesto que este tipo de usuarios realiza muchas menos peticiones al servidor que cualquiera de los otros tipos (únicamente sintoniza un canal de televisión durante un período prolongado de tiempo). Por otro lado, el número de usuarios de tipo `epg_user` puede ser mayor que el número de usuarios `short-lived_user`. Este hecho podría parecer, a simple vista, sorprendente, puesto que los usuarios del tipo `epg_user` realizan muchas más peticiones que los usuarios del tipo `short-lived_user`, el cual está compuesto básicamente por operaciones de inicio y cierre de sesión. Con esta información es posible concluir que estas operaciones de inicio y cierre de sesión son muy costosas en comparación con otras operaciones proporcionadas por el servicio web. También se debe tener en cuenta que, en este caso, se está utilizando un servidor Memcached para almacenar temporalmente datos en memoria para aumentar el rendimiento. Así, mientras que muchas de las operaciones invocadas por los usuarios de tipo `epg_user`, como son la obtención de la lista de canales o la programación de un canal, pueden ser almacenadas en este sistema de caché, no se puede usar este sis-

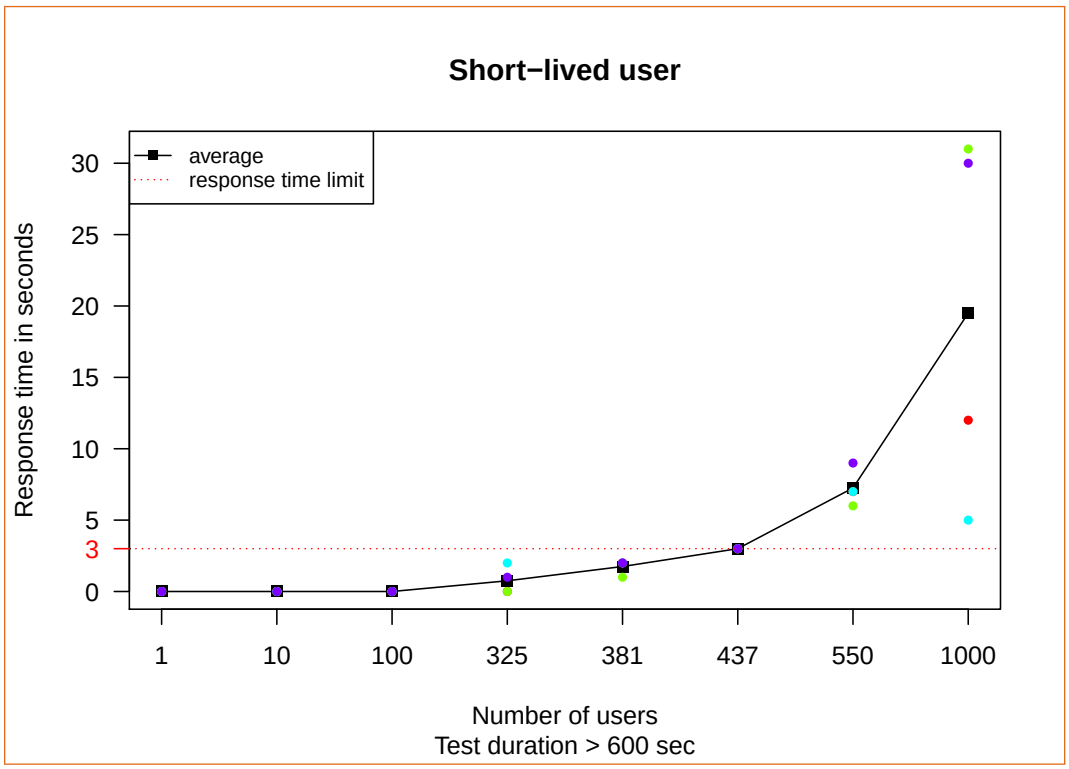


FIGURA 9.6: *Número máximo de usuarios concurrentes del sistema VoDKATV de tipo* `short-lived_user`

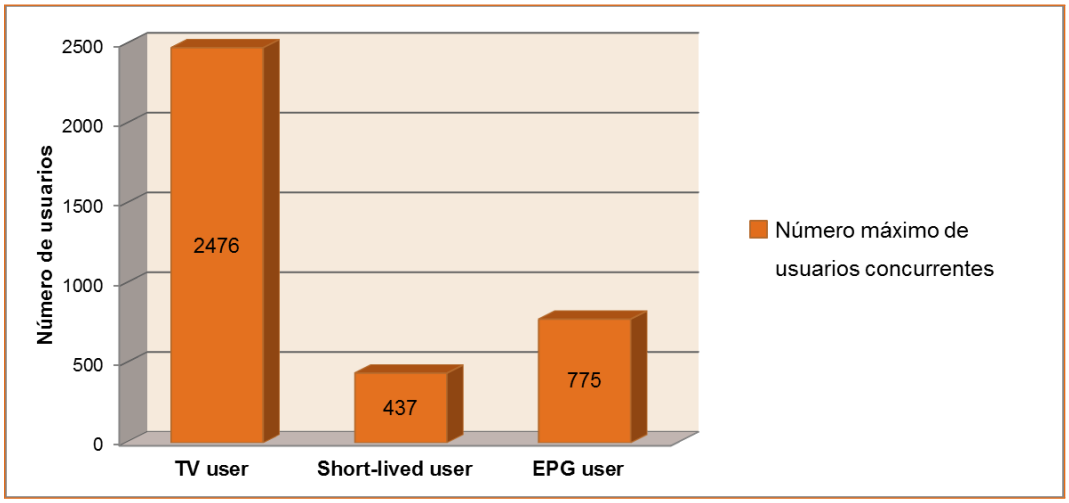


FIGURA 9.7: *Número máximo de usuarios concurrentes del sistema VoDKATV por cada tipo de usuario*

tema para las peticiones de inicio y cierre de sesión.

En la ejecución de esta propiedad se ha observado como el número de pasos necesarios durante la generación y reducción incrementa cuando el número de usuarios soportados en el sistema es mayor. Así, para los usuarios de tipo `tv_user`, para los que se obtiene un valor de 2500 usuarios concurrentes, se han requerido 12 pasos, es decir, 3 pasos más de los que han sido necesarios en la mayoría de los casos comprobados con la primera propiedad. Mientras que ésta puede ser una diferencia sutil en las pruebas funcionales, la gran cantidad de tiempo requerido para ejecutar un único caso de prueba en las pruebas de carga requiere optimizar la generación de estos casos de prueba. Para ello, es posible usar diferentes tipos de generadores. En la siguiente tabla se muestran varias ejecuciones cambiando el generador de datos para el número de usuarios concurrentes, en concreto, usando un generador lineal con aumentos de 300 y 500 usuarios:

Paso	**Paso = 300**	**Paso = 500**
Margen	**Margen = 100**	**Margen = 100**
Usuarios concurrentes	(2475, 2550)	(2500, 2562)
Pasos generación	9	6
Pasos reducción	2	3
Total pasos	11	9

TABLA 9.6: *Número máximo de usuarios concurrentes del sistema VoDKATV usando un contenedor de aplicaciones Apache Tomcat y Memcached con diferentes generadores de datos exponenciales*

Por otro lado, a continuación, se muestra la ejecución de la misma propiedad usando la misma configuración de VoDKATV, pero con generadores exponenciales de base 2 y 10:

Base	**Base = 2**	**Base = 10**
Margen	**Margen = 100**	**Margen = 100**
Usuarios concurrentes	(2432, 2496)	(2476, 2546)
Pasos generación	13	5
Pasos reducción	5	7
Total pasos	18	12

TABLA 9.7: *Número máximo de usuarios concurrentes del sistema VoDKATV usando un contenedor de aplicaciones Apache Tomcat y Memcached con generadores de datos lineales*

Usar un generador u otro depende del dominio concreto en el que se vaya a ejecutar la propiedad. En este caso, un generador lineal con un paso de 500 es el generador que proporciona mejores resultados. Sin embargo, en sistemas que soporten un

gran número de usuarios concurrentes, este generador sería menos eficiente que un generador exponencial de base 10, puesto que, en este último caso, se necesitarían menos pasos para ejecutar las pruebas con un gran número de usuarios. En resumen, la decisión de usar un generador u otro se debería tomar en función del número máximo de usuarios concurrentes para los que el sistema fue diseñado, puesto que éste debería ser similar al obtenido.

9.3.3. Resultados de aplicar la aproximación de pruebas

En esta sección se ha ilustrado cómo formular propiedades no funcionales para probar un sistema software. Así, se ha usado esta aproximación para probar algunos aspectos del sistema VoDKATV, en concreto, la búsqueda del máximo número de usuarios concurrentes soportados, es decir, cuántos usuarios pueden estar usando el servicio web proporcionado por VoDKATV-core simultáneamente sin que la calidad del servicio en términos de latencia se vea afectada.

En esta aproximación, basada en propiedades, los casos de prueba los conforman las diferentes simulaciones realizadas sobre el sistema, las cuales son llevadas a cabo con una cantidad determinada de usuarios concurrentes. Para ello, puesto que esta aproximación se basa en modelar el comportamiento de los usuarios, se debe analizar qué tipo de usuarios usarán el sistema, y qué interacciones provoca cada tipo de usuario con el propio sistema. Con esta información se obtienen una serie de escenarios para la herramienta Megaload, los cuales representan, a través de una especificación escrita en el lenguaje JSON, el comportamiento de los usuarios reales que usarán el sistema VoDKATV.

Con la aproximación basada en propiedades, la ejecución de estos escenarios se realiza automáticamente con diferentes configuraciones. En concreto, se han realizado pruebas con diferente número de usuarios concurrentes y diferente proporción de dichos usuarios en cada escenario. En estas ejecuciones, inicialmente se debe preparar el sistema hasta alcanzar el número de usuarios esperado y, una vez alcanzado, se debe mantener dicho número estable durante un período de tiempo para evaluar el rendimiento del sistema. Aunque se han realizado pruebas variando el período de ejecución, finalmente se ha optado por realizar pruebas de diez minutos de duración. Finalmente, se necesita un tiempo adicional para que los usuarios abandonen el sistema y éste se vuelva a estabilizar.

En total, el tiempo requerido para ejecutar cada uno de los casos de prueba es de aproximadamente 15 minutos. Por tanto, ejecutar todos los casos de prueba necesarios para comprobar una propiedad no funcional puede requerir una cantidad considerable de tiempo. En concreto, puesto que para buscar el máximo número de usuarios soportados por el sistema VoDKATV se ejecutan una media de 8 o 9 casos de prueba, se necesitan aproximadamente 2 horas para encontrar los límites del sistema. Además, puesto que los resultados obtenidos en la ejecución de una propiedad son no deterministas y podrían estar influenciados por agentes ajenos y

no controlados, como pueden ser la carga del servidor o el uso de la red, en este caso se ha ejecutado cada propiedad cinco veces. Esto implica que ejecutar cada propiedad ha requerido aproximadamente 11 horas para su completa ejecución. Es posible que en sistemas más complejos el tiempo para ejecutar este tipo de propiedades se incremente considerablemente, de tal forma que comprobar los requisitos no funcionales de un sistema podría requerir varios días de ejecución. Este tiempo implica tanto la generación del número de usuarios utilizado en cada caso de prueba usando el generador QuickCheck, lo cual es un tiempo comparativamente poco significativo, como la propia ejecución de cada caso de prueba, invocando la secuencia de operaciones especificadas en los escenarios definidos.

Aunque comprobar las propiedades especificadas pueda requerir una gran cantidad de tiempo, la ejecución de las propiedades es una tarea no supervisada, que no requiere interacción humana. Al contrario, usar una aproximación manual requiere que se compruebe el resultado de cada caso de prueba. Por ejemplo, en VoDKATV se requeriría interacción humana cada, aproximadamente, 15 minutos, para preparar y lanzar un nuevo caso de prueba (con una configuración que depende de los resultados obtenidos en ejecuciones pasadas). De esta forma, la persona encargada de ejecutar este tipo de pruebas debe, o bien permanecer esperando hasta que cada caso de prueba termine para poder preparar y lanzar el siguiente manualmente, o bien realizar otro tipo de tareas mientras se ejecuta cada caso de prueba, realizando, por tanto, muchos cambios de contexto entre tareas a lo largo del día (con la consecuente pérdida de tiempo que ello conlleva [364]). En cualquier caso, sea cual sea la opción elegida, la productividad con la aproximación manual decrece con respecto a la aproximación basada en propiedades.

Esta experiencia se corresponde con la realización de pruebas de rendimiento del sistema VoDKATV dentro de la empresa Interoud Innovation antes de la existencia de la herramienta Megaload. En este caso, se habían realizado pruebas de rendimiento con la ayuda de las herramientas Tsung [69] y ApacheBench [65], a través de las cuales ha sido posible obtener datos como el número máximo de usuarios concurrentes soportados por el sistema VoDKATV o cuáles son las operaciones más costosas. Para ello, la aproximación seguida fue la realización de pruebas con diferentes configuraciones, observando manualmente los resultados obtenidos en cada ejecución y repitiendo el proceso múltiples veces hasta conseguir los resultados deseados. Además, la gran cantidad de tiempo requerido para llevar a cabo este tipo de pruebas provoca que éstas no se realicen para todas las versiones del sistema VoDKATV, sino que son ejecutadas cada cierto tiempo. Cabe resaltar que los resultados con la aproximación basada en propiedades son muy similares a los obtenidos anteriormente con la aproximación manual.

La realización de las pruebas usando la herramienta Megaload, además de obtener cuál es el número máximo de usuarios concurrentes soportados por el sistema VoDKATV, así como cierta información acerca de la carga generada en el sistema

por cada uno de los tipos de usuarios que pueden usar VoDKATV, ha revelado la existencia de cuatro situaciones causantes de diversos problemas de rendimiento:

- **Conexiones a la base de datos:** cuando aumenta el número de usuarios (por encima de 600) y las peticiones se vuelven más lentas de atender, el número de conexiones a la base de datos se incrementa considerablemente, causando que todas las conexiones disponibles a la base de datos se agoten y, de esta forma, las nuevas conexiones se rechacen. Estos problemas se mitigan cuando se usan cachés para algunas operaciones costosas, aumentando, de esta forma, el número de usuarios concurrentes soportados, como se muestra en los resultados obtenidos cuando se usa el servidor Memcached.

- **Peticiones para obtener los canales multimedia:** aunque este dato ya se conocía, durante la realización de las pruebas se ha observado que no todas las peticiones a URLs son igual de rápidas. En concreto, las operaciones que devuelven datos sobre canales multimedia o la programación de dichos canales son, de media, hasta tres veces más lentas que el resto de operaciones. Esto es debido a que obtener estos datos del servidor de EPG es un proceso lento. El uso del servidor de caché Memcached mitiga este problema, puesto que se almacenan temporalmente los datos obtenidos del servidor de EPG para que no sea necesario consultarlos cada vez que se necesiten devolver al usuario.

- **Número de usuarios registrados en la base de datos:** se ha encontrado que el tiempo de respuesta de las peticiones al sistema VoDKATV depende del número de usuarios registrados en la base de datos, independientemente del número de usuarios concurrentes que estén usando el sistema. Esta situación ha sido detectada porque para realizar las pruebas de Megaload se han tenido que registrar usuarios ficticios en la base de datos que maneja el componente VoDKATV-core, los cuales serán usados durante las pruebas. Para evitar que durante la realización de las pruebas se utilicen siempre los mismos usuarios para acceder al sistema e invocar las operaciones que éste proporciona, se decidió crear un número lo suficientemente grande de posibles usuarios en la base de datos. Gracias a estas pruebas se comprobó como, por ejemplo, con un número pequeño de usuarios conectados (20 – 40 usuarios), y un gran número de usuarios registrados en base de datos (unos 300000 usuarios), el tiempo de respuesta es mucho mayor, en concreto, 20 veces mayor que con 1000 usuarios registrados en la base de datos.

  Esta situación no se ha convertido en un problema real hasta ahora, puesto que las instalaciones actuales del sistema VoDKATV, es decir, hoteles y pequeños entornos de telecomunicaciones, no tienen un gran número de usuarios registrados en base de datos (siempre menor que 40000 usuarios). En cualquier caso, este potencial problema ha sido mitigado a través de la creación de índices sobre algunas tablas de la base de datos, aunque su análisis ha confirmado que debe realizarse una reestructuración de las tablas que almacenan los

datos de los usuarios y sus preferencias (ordenación personalizada de canales, preferencias de idioma, preferencias de interfaz, etc.) para que el sistema VoDKATV sea completamente escalable en relación al número de usuarios registrados.

- **Almacenamiento de claves de autorización:** durante la ejecución de las pruebas usando propiedades se observó una ligera degradación continua del rendimiento a medida que se ejecutaban las pruebas durante varias semanas. Después de analizar esta situación, se ha observado que su causa es similar al problema descrito anteriormente relacionado con el número de usuarios registrados en el sistema. En este caso, cada vez que un usuario inicia sesión en el sistema, se crea una clave de autorización, que se almacena en base de datos, la cual otorga al usuario el acceso al sistema durante un tiempo. Esta clave se envía como una *cookie* o un *token* cuando se invocan al resto de operaciones del servicio web, y durante la ejecución de las mismas se comprueba si la clave sigue siendo válida para el usuario. Además, la clave de autorización debe ser renovada periódicamente, y se marca como inválida después de invocar la operación de cierre de sesión.

  Después de la ejecución de las pruebas usando Megaload durante varias semanas, el número de claves de autorización era de aproximadamente 2,5 millones. El problema en este caso es que las búsquedas en esta tabla no habían sido optimizadas para ser ejecutadas sobre tal cantidad de tuplas, por lo que la ejecución de las operaciones se convierte en un proceso lento. El uso de índices sobre esta tabla y la mejora de la gestión de las claves ya caducadas (moviéndolas a otras tablas que almacenen históricos) ha sido la solución a este problema.

  Cabe destacar que si no se hubieran realizado este tipo de pruebas durante varias semanas, este problema permanecería oculto hasta que el sistema hubiese sido usado en producción por un gran número de usuarios durante un período largo de tiempo (meses o incluso años). De hecho, en la actualidad, no se conocía de su existencia.

Estas pruebas evalúan el rendimiento del sistema VoDKATV con una configuración de despliegue concreta y con unos servidores físicos concretos. Evidentemente, cuando aparezca un nuevo despliegue a realizar, se pueden tomar estos datos como referencia, aunque si el sistema va a ser instalado en otro tipo de máquinas o con otra configuración diferente, sería recomendable volver a repetir estas pruebas de rendimiento en un entorno similar al que posteriormente será el entorno de producción. De la misma forma, si se prevé que los usuarios usarán el sistema de una manera diferente a la ya especificada, se deberá modelar el comportamiento de estos nuevos tipos de usuarios. De lo contrario, si el comportamiento de los usuarios que usarán el sistema va a ser el mismo que habitualmente, al ya disponer de una implementación de la especificación de dicho comportamiento, así como de las propiedades

definidas para obtener el número máximo de usuarios concurrentes, únicamente se debería configurar el sistema en un entorno similar al entorno de producción y volver a ejecutar dichas propiedades para analizar los resultados obtenidos.

Por último, cabe destacar que en las pruebas no funcionales, al igual que ocurre para las pruebas funcionales, escribir propiedades es más difícil que simplemente ejecutar ejemplos concretos. Por esta razón, obtener una propiedad que describa exactamente el comportamiento a probar es lo más costoso de este tipo de aproximaciones, puesto que su ejecución, como se ha comentado anteriormente, aunque puede requerir una gran cantidad de tiempo, es una tarea que no requiere atención humana. No obstante, las propiedades no funcionales a comprobar, como es la obtención del número máximo de usuarios concurrentes, suelen ser comunes en varios sistemas software. Por esta razón, se ha incluido soporte implícito para este tipo de propiedades dentro de la herramienta Megaload, de tal forma que la persona que las quiera utilizar no tenga que preocuparse por los detalles de las mismas, como son la implementación de los generadores o la estrategia de reducción asociada a los mismos.

9.4. Resumen

En este capítulo se explica cómo usar una aproximación basada en propiedades para realizar pruebas no funcionales. Así, esta aproximación se ha utilizado para encontrar los límites de un sistema software, en concreto, un servicio web proporcionado por el sistema VoDKATV, en función del número máximo de usuarios concurrentes soportados. Para ello, se ha usado como criterio de aceptación el tiempo de respuesta medio requerido para obtener el resultado de las operaciones invocadas de dicho servicio web.

La ejecución de las pruebas se ha llevado a cabo utilizando la herramienta Megaload, la cual permite ejecutar pruebas de rendimiento de sistemas software. Esta herramienta se integra con QuickCheck, que se usa como lenguaje de especificación de propiedades no funcionales a partir de las cuales se generan y ejecutan casos de prueba concretos. Al contrario que las pruebas funcionales, las pruebas de rendimiento normalmente requieren una gran cantidad de tiempo (varias horas o incluso días) para su ejecución, y comprueban algún tipo de comportamiento no determinista. Por tanto, los generadores de datos usados en dichas propiedades deben tener en cuenta estos aspectos.

Si bien es cierto que los ejemplos mostrados en este capítulo son casos sencillos, éstos muestran cómo el uso de esta aproximación puede suponer un ahorro considerable de tiempo con respecto a otras estrategias manuales debido a la ejecución desatendida, y, gracias a esto, por la mejor utilización del tiempo disponible para estas pruebas y la detección de problemas.

No obstante, tomando como base la aproximación descrita en este capítulo, existen varias líneas de investigación abiertas a considerar:

- Por un lado, se deben tener en cuenta más aspectos del sistema, además del tiempo de respuesta, el número de peticiones rechazadas o las peticiones no atendidas a tiempo, como es la monitorización de los servidores donde se ejecuta el sistema a probar, usando parámetros como el uso del procesador, conexiones de red, uso de red, uso de entrada/salida, o número de ficheros abiertos, entre otros.
- Por otro lado, se debe estudiar cómo formular propiedades más complejas, que dependan de más de un parámetro, cómo puede ser, en el ejemplo de VoDKATV, buscar qué distribuciones de tipos de usuarios podrán ser soportadas por el sistema VoDKATV y cuales causarían un mal comportamiento.
- Finalmente, el uso de una aproximación basada en propiedades para probar sistemas distribuidos, compuestos por múltiples nodos que aportan una alta disponibilidad al sistema, es otro de los casos donde podría ser posible formular propiedades. En este último caso, estas propiedades se encargarían de comprobar el funcionamiento del sistema cuando alguno de los nodos deja de estar disponible o vuelve a estar disponible de nuevo.

10

Conclusiones

10.1. Introducción

Cuando un proyecto software es concebido y se toma la decisión de desarrollar un sistema, son necesarias unas fases iniciales de análisis y diseño, en las que se especifica la arquitectura del sistema, describiendo a alto nivel qué componentes forman parte del mismo y las interacciones entre ellos. Durante el ciclo de vida del producto software, las tareas de pruebas son fundamentales para evaluar el desarrollo de dichos componentes y dichas interacciones. Así, en este trabajo, se describen una serie de metodologías y técnicas que pueden ser utilizadas para la realización de las pruebas del software en diferentes niveles. De esta forma, las metodologías y técnicas desarrolladas pueden ser usadas para probar los componentes de forma individual (tanto los módulos unitarios que forman parte de los mismos, como las APIs de integración que proporcionan), las integraciones entre ellos, así como algunos aspectos del sistema completo integrado, como es el rendimiento del sistema.

Una de las características principales de las técnicas de pruebas desarrolladas es que todas ellas se basan en el uso de propiedades para describir el comportamiento del software a probar. Estas propiedades se formulan utilizando el lenguaje de programación funcional Erlang, cuya naturaleza funcional y su sintaxis declarativa permite especificarlas de una forma natural y concisa. Consecuentemente, se ha utilizado la versión Erlang de QuickCheck como herramienta de pruebas basada en propiedades. Cabe destacar que Erlang y QuickCheck han sido usados para realizar las pruebas en todos los niveles de pruebas, siendo, por tanto, posible usar un formalismo y herramienta únicos. Además, a pesar de que el lenguaje de especificación de pruebas es Erlang, este hecho no ha restringido usar esta aproximación a la hora

de probar componentes implementados en diferentes tecnologías.

Con respecto al nivel unitario, se ha descrito cómo combinar las pruebas basadas en propiedades con metodologías de desarrollo ágiles, en concreto, el desarrollo dirigido por las pruebas, obteniendo como resultado una nueva metodología a la que se ha referido en este trabajo como *desarrollo dirigido por propiedades*. Esta metodología, basada en la definición de propiedades que especifican el comportamiento a probar antes que la propia implementación de dicho comportamiento, puede ser empleada durante la implementación de los módulos unitarios.

Los módulos unitarios se agrupan en componentes que forman parte del sistema software. Así, también es importante realizar las pruebas a nivel de componente. Desde este punto de vista, se ha desarrollado una metodología de pruebas, que describe los pasos a realizar y los roles de las personas involucradas durante la implementación y pruebas de APIs de integración. En este caso, usando una aproximación basada en propiedades, se debe escribir un modelo que especifique las operaciones que forman parte de la API de integración a probar, junto con las precondiciones y postcondiciones de cada una de ellas. A partir de esta especificación, la herramienta de pruebas basada en propiedades se encarga de generar secuencias aleatorias de llamadas a operaciones basándose en las precondiciones especificadas, y comprobando las postcondiciones asociadas a cada una de ellas. Uno de los puntos principales de esta aproximación es que la misma especificación de pruebas es válida para probar diferentes implementaciones de una misma API de integración, únicamente cambiando los adaptadores empleados para conectar la ejecución de los casos de prueba generados con la implementación concreta a probar. Además, se ha realizado una especialización de esta técnica cuando la API de integración es un servicio web, prestando especial atención a la evolución del código de pruebas cuando el servicio web se modifica a lo largo del tiempo.

Como la aproximación desarrollada para probar el software a nivel de componente se trata de una técnica de pruebas de caja negra, que no tiene en cuenta los detalles internos del sistema a probar, es posible utilizar un modelo de la API de integración a probar para generar automáticamente esta especificación en forma de propiedades. De esta forma, se ha desarrollado una herramienta que permite obtener una especificación de pruebas QuickCheck a partir de un modelo UML, complementado con restricciones OCL, que describe la API de integración a probar. Cuando la API de integración es un servicio web, el modelo UML se reemplaza por una especificación WSDL, más natural cuando se trata de servicios web.

Para el nivel de integración también se han realizado contribuciones en cuanto a las pruebas del software. En este caso, el objetivo principal de este tipo de pruebas es probar que la integración entre dos componentes funciona de la manera deseada, es decir, que se realizan las llamadas esperadas a un componente externo en el momento esperado y con los argumentos esperados, y se obtiene el valor devuelto por este componente externo satisfactoriamente. Para ello, al igual que en el resto

de niveles de pruebas del software, se ha propuesto el uso de las pruebas basadas en propiedades. Sin embargo, en este caso, la técnica desarrollada se basa en el reemplazo del componente con el que se integra el propio componente a probar por otro componente más sencillo que proporciona una API de integración equivalente, a partir del cual se pueden inspeccionar las trazas generadas durante la comunicación entre ambos y, de esta forma, determinar, con las propiedades formuladas, si la integración entre los dos componentes se ha implementado de manera apropiada.

Por otro lado, a nivel de sistema, se han establecido las bases para usar las pruebas basadas en propiedades para la realización de las pruebas de rendimiento. En este caso, se ha argumentado la necesidad de definir generadores de datos propios en QuickCheck para realizar este tipo de pruebas de manera eficiente, puesto que los generadores proporcionados por defecto no son adecuados para llevar a cabo este tipo de pruebas.

Además, durante la realización de este trabajo también se han implementado algunas herramientas necesarias para llevar a cabo las metodologías desarrolladas.

10.2. Contribuciones

De modo resumido, estas son las principales contribuciones de este trabajo:

- En relación con las pruebas de unidad:
  - Desarrollo basado en propiedades: metodología para la realización de las pruebas de unidad de los módulos que forman parte de un componente de software, integrada con las actividades de implementación de dichos módulos, a través de una aproximación basada en propiedades [311].
- En relación con las pruebas de componente:
  - Metodología para la realización de pruebas de APIs de integración, integrada en el proceso de definición e implementación de dichas APIs de integración [136, 187, 189].
  - Aproximación para realizar pruebas de servicios web, basada en la metodología definida para probar APIs de integración, y automatizando algunas partes de dicho proceso, en concreto, la generación de un esqueleto para la máquina de estados QuickCheck y la generación automática de adaptadores a partir de una especificación WSDL del servicio web a probar [190, 247, 250, 253].
  - Aproximación basada en el uso de la herramienta Wrangler para ayudar a modificar el código de pruebas a medida que la implementación del servicio web evoluciona [247].

- Librería para la implementación de adaptadores Java que conecten los casos de prueba con el sistema a probar, cuando éste es una API de integración genérica: https://github.com/miguelafr/mbt_erlang_adapter.
- Herramienta de generación de propiedades y máquinas de estados QuickCheck a partir de una especificación UML y OCL: https://github.com/miguelafr/umlocl2eqc.
- Herramienta de generación de propiedades y máquinas de estados QuickCheck a partir de una especificación WSDL y OCL: https://github.com/hferreiro/WSDL-OCLToErlang.
- Herramienta de generación de conectores Erlang que invocan las operaciones definidas en una especificación WSDL: https://github.com/RefactoringTools/WSToolkit.
- Herramienta de generación de esqueletos de máquinas de estados QuickCheck (generadores y cabeceras de funciones) a partir de una especificación WSDL: https://github.com/RefactoringTools/WSToolkit.

- En relación con las pruebas de integración:
  - Metodología para la realización de pruebas de integración [137–139].
- En relación con las pruebas de sistema:
  - Aproximación basada en propiedades para la realización de pruebas de rendimiento de servicios web [188].

De esta forma, este trabajo ofrece una serie de metodologías de pruebas, integradas en el proceso de desarrollo, las cuales se basan en aproximaciones de pruebas automáticas basadas en propiedades. Además, se han proporcionado las herramientas necesarias para llevar a cabo estas metodologías, las cuales permiten llevar a la práctica los conceptos explicados.

Finalmente, cabe mencionar también que el uso de todas las metodologías y técnicas de pruebas desarrolladas se ha ilustrado a través de un sistema software real, llamado VoDKATV, en concreto:

- Se ha descrito, utilizando un módulo del sistema VoDKATV, cómo implementar dicho módulo cuando el desarrollo está dirigido por propiedades.
- Se ha ilustrado como probar APIs de integración y servicios web proporcionados por componentes que forman parte de este sistema.
- Se ha probado la integración entre dos componentes integrantes del sistema VoDKATV.

- Y, finalmente, se ha ilustrado cómo usar propiedades para realizar pruebas de rendimiento en este sistema.

10.3. Lecciones aprendidas

El reconocimiento de la importancia que tienen las tareas de pruebas dentro del ciclo de vida del software es cada vez mayor. Sin embargo, paradójicamente, en muchas ocasiones, éstas son omitidas o no realizadas con toda la rigurosidad necesaria, lo cual normalmente se intenta justificar a través de la presión de entregar un producto a tiempo al cliente. La complejidad de realizar las pruebas, el tiempo y los recursos que normalmente se necesitan para llevarlas a cabo, el desconocimiento de las técnicas adecuadas, el uso inadecuado de las herramientas de pruebas, o incluso la inexistencia de herramientas que realmente se adapten a las peculiaridades de algunos sistemas son algunos de los motivos por lo que esto realmente ocurre.

Como se puede intuir, no realizar las pruebas del software no es una solución para solventar este problema. Las contribuciones de esta tesis tratan de facilitar esta tarea. Para ello, se proporcionan metodologías y técnicas, junto con las herramientas necesarias, adecuadas para los diferentes niveles de pruebas del software, con el fin de que sean utilizadas durante el ciclo de vida de un producto para realizar las pruebas del mismo. Así, aunque las pruebas automáticas requieren un esfuerzo necesario mayor en el diseño e implementación de los casos de prueba comparado con una aproximación de pruebas no automática, la automatización de las pruebas trae consigo una serie de beneficios asociados, como son la reusabilidad de las pruebas, la repetibilidad de las mismas, el menor esfuerzo necesario o la menor supervisión constante para su ejecución.

Por otra parte, como se ha descrito a lo largo de los capítulos del presente trabajo, las aproximaciones de pruebas propuestas están basadas en el uso de propiedades. Durante la realización de esta tesis se ha observado cómo este tipo de aproximaciones pueden ser inicialmente más costosas que las aproximaciones tradicionales en las que los casos de prueba se escriben manualmente uno a uno. Esto es debido a la dificultad que suele suponer formalizar las propiedades que describen el comportamiento del sistema a probar. En contrapartida, estas aproximaciones suelen ser más eficientes y eficaces que las aproximaciones tradicionales una vez superada esta barrera inicial.

Con respecto a la eficiencia, si se tiene en cuenta el tiempo dedicado a la realización de las pruebas, éste suele ser menor que en las aproximaciones tradicionales, como se ha mostrado en la realización de las pruebas de rendimiento. También se incrementa la eficiencia si se considera el aumento en el número de casos de prueba automáticamente generados y ejecutados cuando se introduce el paradigma de las pruebas basadas en propiedades. Así, el uso de las aproximaciones desarrolladas

también ha revelado que suele ser necesario mucho menos código fuente de pruebas que en las aproximaciones en las que los casos de prueba son especificados manualmente. Por tanto, como los casos de prueba son generados automáticamente a partir de las propiedades que describen el comportamiento del sistema a probar, se ha conseguido el objetivo de escribir menos código y, al mismo tiempo, realizar pruebas más exhaustivas. Además, este proceso de generación automática de casos de prueba a partir de una serie de generadores de datos no conlleva una degradación significativa del rendimiento en la ejecución de las pruebas del software. Por otro lado, la valoración de las aproximaciones de pruebas desarrolladas con respecto a la eficacia de las mismas también es positiva, puesto que su uso para probar un sistema real, como es VoDKATV, ha servido para detectar defectos que no habían sido detectados anteriormente.

A pesar de las ventajas que ofrecen las aproximaciones basadas en propiedades, su adopción dentro de un equipo de trabajo es más complicada de lo que a priori podría parecer, sobre todo al combinarlas con el desarrollo dirigido por las pruebas. Por una parte, es necesaria una preparación previa para el uso de estas nuevas aproximaciones y, en muchos casos, las personas se muestran reacias al uso de nuevos métodos para llevar a cabo el trabajo diario. Así, para usar de manera habitual una metodología como el desarrollo dirigido por las pruebas y, en concreto, usando propiedades que describan el comportamiento del software a probar, se han necesitado meses de adaptación dentro del equipo de desarrollo de la compañía LambdaStream, durante los cuales ha sido discutido en multitud de ocasiones internamente la eficiencia y eficacia de este tipo de aproximaciones, para que finalmente pudieran ser apreciadas las ventajas que realmente ofrecen. Por esta razón, se ha visto cómo una curva de aprendizaje fácil es uno de los aspectos más importantes para la adopción de las nuevas técnicas, metodologías y herramientas desarrolladas.

Relacionado con el punto anterior, cabe mencionar la utilización del lenguaje de programación Erlang como lenguaje de especificación de pruebas. Durante la realización de esta tesis, dentro del equipo de desarrollo de Interoud Innovation (y, anteriormente, LambdaStream) se encontraban dos equipos diferenciados. Uno de estos equipos se encargaba del desarrollo de la capa *middleware* del *set-top-box*, para la cual se empleaba mayoritariamente el propio lenguaje de programación Erlang. Por otro lado, el otro equipo se encargaba del desarrollo de las aplicaciones cliente-servidor, usando Java como lenguaje de programación de la parte servidor y, normalmente, tecnologías web como HTML, CSS y JavaScript para la parte cliente. Como es de esperar, la adopción del uso de la versión Erlang de QuickCheck para realizar las pruebas basadas en propiedades fue mucho más fácil para los integrantes del primer equipo de trabajo. Para las personas del segundo equipo, el uso de Erlang supuso una barrera inicial importante para el uso de estas técnicas.

Existen algunas situaciones, como son, por ejemplo, probar una API de integración, el caso particular de un servicio web, o realizar las pruebas de rendimiento,

que han sido tratadas desde un punto de vista de caja negra en este trabajo. Por tanto, en estos casos, no es necesario conocer los detalles internos del sistema a probar para realizar este tipo de pruebas. Esto provoca que estas pruebas puedan ser llevadas a cabo por un equipo de probadores especializados en la realización de las pruebas del software. Aunque este equipo no existe en Interoud Innovation, es muy probable que, en este caso, el uso de un lenguaje de especificación nuevo como lenguaje de pruebas no sea un problema. Al contrario, en este caso, podría ser una ventaja, puesto que este lenguaje, es decir, Erlang, se utilizaría de manera uniforme para realizar las pruebas del software en diferentes niveles.

Otro de los aspectos a mencionar es que el simple uso de una aproximación de pruebas u otra no es suficiente para lograr buenos conjuntos de prueba. Así, durante las etapas iniciales de la adopción de las pruebas basadas en propiedades dentro del equipo de desarrollo de LambdaStream, el uso de revisiones de código ha servido para detectar propiedades que no generaban casos de prueba adecuados para probar un comportamiento determinado, o no se realizaban todas las comprobaciones necesarias. Así, en muchas ocasiones, no es la técnica, sino la experiencia de la persona que utilice dicha técnica, la que va a condicionar los resultados de las pruebas.

De la misma forma, en algunas ocasiones, se ha observado cómo la intuición de la persona que esté realizando las pruebas puede llevar a querer usar un caso de prueba concreto, o cómo para algunas situaciones podría ser más adecuado realizar las pruebas con valores fijos conocidos de antemano. En estos casos, escribir una propiedad que genere este caso de prueba (o un conjunto de casos de prueba prefijados de antemano) puede resultar más difícil que usar una aproximación en la que dicho caso de prueba se escriba manualmente. Obviamente, aunque por norma general, en este trabajo se recomienda el uso de una aproximación basada en propiedades para especificar el comportamiento a probar de un componente software, esto no es incompatible con el uso de casos de prueba especificados manualmente para este tipo de situaciones.

Finalmente, destacar que durante la realización de estas tesis, las metodologías y técnicas de pruebas desarrolladas han sido utilizadas tanto con ejemplos sencillos que han ayudado a la elaboración de las mismas, como con un caso real, en concreto, el sistema VoDKATV. En general, mientras que el uso de estas aproximaciones con ejemplos sencillos normalmente muestra los beneficios de las mismas en comparación con otras aproximaciones tradicionales; usarlas con el sistema VoDKATV ha servido normalmente para encontrar limitaciones de las mismas, debidas a que la complejidad de este sistema, muchas veces, ha supuesto encontrar situaciones que no se habían tenido en cuenta durante la definición inicial de estas aproximaciones. De esta forma, usar diferentes partes del sistema VoDKATV, además de permitir ilustrar el funcionamiento de las metodologías y técnicas desarrolladas con un caso real, ha servido para mejorar las aproximaciones iniciales de las mismas, permitiendo que éstas puedan ser fácilmente configuradas y ajustadas para funcionar en

diferentes entornos de desarrollo.

10.4. Líneas de investigación abiertas

Las metodologías y técnicas que se han desarrollado en este trabajo pueden ser empleadas en multitud de etapas en el desarrollo de un sistema software, desde la implementación de los componentes individuales hasta las pruebas del sistema completo. No obstante, los resultados obtenidos con la realización de esta tesis abren nuevas líneas de investigación a explorar.

Por un lado, un aspecto que merece la pena destacar aquí es el mantenimiento de las pruebas. En algunas situaciones, cuando la implementación de un componente evoluciona, los programadores optan por borrar un caso de prueba que no funciona, en vez de analizar y arreglar las pruebas ya existentes. Esto es debido a que el código de pruebas suele convertirse en algo difícil de mantener y evolucionar a medida que cambia la implementación del sistema a probar. Aunque en este trabajo se han descrito aproximaciones para realizar esta tarea, en concreto, para las pruebas de servicios web, sería interesante investigar este aspecto en otras situaciones y de una manera genérica.

Por otro lado, en el caso concreto de las pruebas de unidad, la metodología propuesta ha sido ejemplificada a través de la implementación de un módulo Erlang, realizando las pruebas correspondientes con la versión Erlang de QuickCheck. Aunque el lenguaje de programación Erlang ofrece interfaces para comunicarse con otros lenguajes de programación, como Java o C, sería interesante investigar cómo aplicar esta aproximación, puramente funcional y basada en propiedades, para implementar código en un lenguaje de programación no funcional, analizando las ventajas e inconvenientes con respecto a una aproximación más tradicional, basada en la especificación de casos de prueba concretos.

Con respecto a las pruebas de APIs de integración, cabe recordar que la infraestructura de pruebas necesaria para las mismas se inspira en la arquitectura de TTCN-3, puesto que, de esta forma, se separa la especificación de pruebas de la implementación de la API de integración a probar. Aunque se ha propuesto el uso de propiedades para esta especificación de pruebas, el uso del lenguaje TTCN-3 constituye una aproximación estándar, y es usada en muchos ámbitos para la realización de pruebas de componentes. Por tanto, una posible vía a explorar es la generación automática de conjuntos de prueba TTCN-3 a partir de una especificación en el lenguaje QuickCheck. Otra posible línea de investigación, y más interesante, es realizar lo contrario, es decir, generalizar un conjunto de pruebas escritas en el lenguaje TTCN-3 para obtener un conjunto de propiedades QuickCheck, de la misma forma que ya se hizo para otro tipo de situaciones similares, como es la obtención de una especificación de pruebas QuickCheck a partir de un conjunto de pruebas EUnit [91, 92, 246] o JUnit [233, 234].

Continuando con las pruebas de servicios web, cabe mencionar que la aproximación de pruebas propuesta se basa en el uso de una especificación WSDL, la cual se complementa con una serie de precondiciones y postcondiciones asociadas a cada operación, escritas en QuickCheck o en OCL. Sin embargo, puede ser conveniente especializar este tipo de pruebas para los servicios web basados en el estilo de arquitectura REST, los cuales siguen una serie de convenciones específicas para la invocación de las operaciones que forman parte de los mismos. Así, sería interesante investigar cómo el uso de estas convenciones afecta a la especificación del servicio web, puesto que es probable que la especificación de pruebas y, en general, la arquitectura necesaria para llevarlas a cabo, sea más simple en este caso.

Para las pruebas de integración, por su parte, también existen líneas de investigación abiertas. Así, al igual que para probar APIs de integración o servicios web se han proporcionado herramientas que permiten generar propiedades que describen el comportamiento del sistema a probar a partir de una especificación más formal, usando UML y OCL, o WSDL y OCL respectivamente, podría facilitarse esta opción para la realización de pruebas de integración. En este caso, una posible alternativa es el uso de diagramas de secuencia UML que describan las interacciones entre componentes software de un sistema. Estos diagramas muestran ejemplos de funcionamiento, por lo que, una posible opción es obtener una especificación basada en propiedades a partir de la generalización de los ejemplos descritos en uno o varios diagramas de secuencia, de una forma similar a la comentada anteriormente para la generalización de conjuntos de pruebas escritas en el lenguaje TTCN-3.

Otro aspecto en el que es posible profundizar es la realización de pruebas no funcionales usando una aproximación basada en propiedades. Así, se ha mostrado una aproximación inicial que muestra cómo podrían enfocarse las pruebas de rendimiento usando propiedades, las cuales describen el comportamiento no funcional esperado de un sistema software en términos de rendimiento del mismo. Sin embargo, una línea de investigación a explorar es cómo escribir propiedades específicas para probar arquitecturas distribuidas, en las que existen varios nodos replicados realizando las mismas funciones, y en los que es interesante comprobar diferentes aspectos de rendimiento en cuanto al número de nodos disponibles, la caída de un nodo, o que un nodo nuevo esté disponible de nuevo, entre otros.

Finalmente, existen una serie de líneas abiertas para investigar el uso de las pruebas basadas en propiedades para probar partes específicas de un sistema. Un ejemplo son las interfaces de usuario, las cuales, en la actualidad, son mayormente probadas con herramientas de *captura y repetición*, es decir, se captura un uso concreto de la interfaz de usuario, y esta captura se repite posteriormente para comprobar si la interfaz de usuario se sigue comportando de la misma manera. Aplicar las pruebas basadas en propiedades para probar interfaces de usuario podría suponer una mejora significativa en la realización de este tipo de pruebas.

En conclusión, las tareas de pruebas son esenciales dentro del desarrollo de soft-

ware, y llevarlas a cabo a través de una aproximación basada en propiedades tiene una serie de ventajas asociadas. De esta forma, utilizar las metodologías y técnicas de pruebas desarrolladas en este trabajo permiten realizar las pruebas del software de una manera eficiente y eficaz. Por esta razón, desarrollar nuevos métodos de pruebas basados en el uso de propiedades, que estén especializados para probar diferentes tipos de aplicaciones software específicas o aspectos concretos de sistemas software, permitirá ofrecer las ventajas de las aproximaciones de pruebas basadas en propiedades en más situaciones de las consideradas en este trabajo, ayudando a incrementar la calidad de los productos software.

ÍNDICE

Q

R

S

T

U

V

W

X

Acrónimos

API Application Programming Interface.

ASCII American Standard Code for Information Interchange.

ASD Adaptative Software Development.

ATS Abstract Test Suite.

ATSC Advanced Television System Committee.

BBC British Broadcasting Corporation.

BNF Backus Normal Form.

BNFC BNF Converter.

BOSH Bidirectional-streams Over Synchronous HTTP.

BSS Business Support Systems.

CAS Conditional Access Systems.

CMMI Capability Maturity Model Integration.

CPM Category-Partition Method.

CSS Cascading Style Sheets.

CSV Comma-separated Values.

DASH Dynamic Adaptive Streaming over HTTP.

DFT Design for Testability.

DOC Depended-on Component.

DRM Digital Rights Management.

DSDM Dynamic Systems Development Method.

DSL Domain Specific Language.

DVB Digital Video Broadcasting.

EPG Electronic Program Guide.

ETS Executable Test Suite.

ETSI European Telecommunications Standards Institute.

FDD Feature Driven Development.

FP7 7th Framework Programme for Research and Technological Development.

FSM Finite State Machine.

FTP File Transfer Protocol.

HED Home Entertainment Device.

HLS HTTP Live Streaming.

HTML HyperText Markup Language.

HTTP Hypertext Transfer Protocol.

HTTPS Hypertext Transfer Protocol Secure.

IDE Integrated Development Environment.

IEEE Institute of Electrical and Electronics Engineers.

IETF Internet Engineering Task Force.

IP Internet Protocol.

IPTV Internet Protocol Television.

ISDB Integrated Services Digital Broadcasting.

ISO International Organization for Standardization.

J2EE Java Platform Enterprise Edition.

JSON JavaScript Object Notation.

LCD Liquid Crystal Display.

LED Light-Emitting Diode.

LOC Lines of Code.

MADS Models and Applications of Distributed Systems.

MC/DC Modified Condition/Decision Coverage.

MHP Multimedia Home Platform.

MPEG-2 Moving Pictures Experts Group 2.

MPEG-4 Moving Pictures Experts Group 4.

NFS Network File System.

NIST National Institute of Standards and Technology.

NTSC National Television System Committee.

OAT Orthogonal Array Testing.

OCL Object Constraint Language.

OMG Object Management Group.

OSI Open Systems Interconnection.

OSS Operations Support Systems.

OTAN Organización del Tratado del Atlántico Norte.

OTT Over the Top.

PAL Phase Alternating Line.

PBT Property-Based Testing.

PDD Property-Driven Development.

PVR Personal Video Recorder.

RAD Rapid Application Development.

REST Representational State Transfer.

RTSP Real Time Streaming Protocol.

SDK Software Development Kit.

SECAM Séquentiel Couleur à Mémoire.

SEI Software Engineering Institute.

SMTP Simple Mail Transfer Protocol.

SOA Service-Oriented Architecture.

SOAP Simple Object Access Protocol.

SOC Service-Oriented Computing.

SPE Software Performance Engineering.

SQL Structured Query Language.

SUT System under Test.

SWRL Semantic Web Rule Language.

TCP Transmission Control Protocol.

TDD Test-Driven Development.

TE Test Execution.

TSL Test Specification Language.

TTCN Testing and Test Control Notation.

TTCN-1 Testing and Test Control Notation Version 1.

TTCN-2 Testing and Test Control Notation Version 2.

TTCN-3 Testing and Test Control Notation Version 3.

UDP User Datagram Protocol.

UML Unified Modeling Language.

URL Uniform Resource Locator.

UTP UML Testing Profile.

VoDKA Video on Demand Kernel Architecture.

VoDKATV Video on Demand Kernel Architecture Television.

WADL Web Application Description Language.

WSDL Web Services Description Language.

WSDL-S Web Services Semantics.

XML eXtensible Markup Language.

XMPP Extensible Messaging and Presence Protocol.

XP eXtreme Programming.

XSD XML Schema Definition.

BIBLIOGRAFÍA

[1] IEEE Standard Glossary of Software Engineering Terminology. *IEEE Std 610.12-1990*, pages 1–84, IEEE, Dec. 1990.

[2] SWRL – Semantic Web Rule Language. http://www.w3.org/Submission/SWRL, May 2004.

[3] IEEE Standard Classification for Software Anomalies. *IEEE Std 1044-2009 (Revision of IEEE Std 1044-1993)*, pages 1–23, IEEE, Jan. 2010.

[4] PROWESS project – Property-Based Testing for Web Services. http://www.prowessproject.eu/, 2012-2015.

[5] Amino – IPTV Technology, Solutions & Streaming. http://www.aminocom.com/, Apr. 2015.

[6] Apache Axis2. http://axis.apache.org/, Apr. 2015.

[7] Apache CXF – An Open-Source Services Framework. http://cxf.apache.org/, Apr. 2015.

[8] Apache JMeter. http://jmeter.apache.org/, Apr. 2015.

[9] Apache Tomcat. http://tomcat.apache.org/, Apr. 2015.

[10] AT&T Uverse. http://www.att.com/shop/u-verse.html, Apr. 2015.

[11] BBC. http://www.bbc.com/, Apr. 2015.

[12] BBC iPlayer. http://www.bbc.co.uk/iplayer, Apr. 2015.

[13] BNFC – The BNF Converter. http://bnfc.digitalgrammars.com/, Apr. 2015.

[14] BTT – Broadbit Test Tool. http://www.broadbit.com/page8/page2/page2.html, Apr. 2015.

[15] Cisco MediaHighway. http://www.cisco.com/c/en/us/products/video/mediahighway/index.html, Apr. 2015.

[16] CppUnit. http://cppunit.sourceforge.net/, Apr. 2015.

[17] CUnit. http://cunit.sourceforge.net/, Apr. 2015.

[18] Devoteam GmbH – TTCN-3 Toolbox. http://www.devoteam.de/en/themen/testing/ttcn-test-solutions-and-beyond/, Apr. 2015.

[19] DVB – Digital Video Broadcasting. https://www.dvb.org/, Apr. 2015.

[20] DVB Standards. https://www.dvb.org/standards, Apr. 2015.

[21] Eclipse. https://eclipse.org/, Apr. 2015.

[22] Elvior – TestCast. http://www.elvior.com/testcast/introduction, Apr. 2015.

[23] Erlang gen_server. http://www.erlang.org/doc/man/gen_server.html, Apr. 2015.

[24] Erlsom – An Erlang Libary for XML Parsing. http://sourceforge.net/projects/erlsom/, Apr. 2015.

[25] ETSI – European Telecommunications Standards Institute. http://www.etsi.org/, Apr. 2015.

[26] EUnit – a Lightweight Unit Testing Framework for Erlang. http://www.erlang.org/doc/apps/eunit/chapter.html, Apr. 2015.

[27] EUnit Reference Manual. http://erlang.org/doc/apps/eunit/, Apr. 2015.

[28] EWE AG – Anbieter für Strom, Erdgas, DSL & Mobilfunk aus einer Hand. http://www.ewe.de/, Apr. 2015.

[29] GNU Emacs – GNU Project – Free Software Foundation (FSF). http://www.gnu.org/software/emacs/, Apr. 2015.

[30] Google Analytics. http://www.google.com/analytics/, Apr. 2015.

[31] HttpUnit. http://httpunit.sourceforge.net/, Apr. 2015.

[32] HUnit. http://hunit.sourceforge.net/, Apr. 2015.

[33] IEEE Computer Society. http://www.computer.org/, Apr. 2015.

[34] Interoud Innovation. http://www.interoud.com/, Apr. 2015.

[35] IPTV Cloud Middleware. http://www.interoud.com/index.php?page=iptv-cloud-middleware&hl=en_UK, Apr. 2015.

[36] Irdeto. http://www.irdeto.com/, Apr. 2015.

[37] ISO – International Organization for Standardization. http://www.iso.org/, Apr. 2015.

[38] Jinterface Reference Manual. http://www.erlang.org/doc/apps/jinterface/index.html, Apr. 2015.

[39] JSON Schema and Hyper-Schema. http://json-schema.org/, Apr. 2015.

[40] JUnit. http://junit.org/, Apr. 2015.

[41] LoadRunner. http://www8.hp.com/us/en/software-solutions/loadrunner-load-testing/, Apr. 2015.

[42] MADS – Models and Applications of Distributed Systems Group. http://www.madsgroup.org/, Apr. 2015.

[43] Memcached. http://memcached.org/, Apr. 2015.

[44] Millennium Hotel Salmiya. http://www.johndavidedison.com/#!salmiya/cey7, Apr. 2015.

[45] Nagra Digital Television. http://dtv.nagra.com/, Apr. 2015.

[46] Netflix. https://www.netflix.com/, Apr. 2015.

[47] nginx. http://nginx.org/, Apr. 2015.

[48] NUnit. http://www.nunit.org/, Apr. 2015.

[49] OCLNL – A tool for informal and formal requirements specifications. http://www.key-project.org/oclnl/, Apr. 2015.

[50] OMG – Object Management Group. http://www.omg.org/, Apr. 2015.

[51] OMG Formal Specifications. http://www.omg.org/spec/, Apr. 2015.

[52] OpenTTCN. http://www.openttcn.com/, Apr. 2015.

[53] Parasoft Corporation. http://www.parasoft.com/, Apr. 2015.

[54] Piwik – Free Web Analytics Software. http://piwik.org/, Apr. 2015.

[55] PostgreSQL. http://www.postgresql.org/, Apr. 2015.

[56] PyUnit – the standard unit testing framework for Python. http://pyunit.sourceforge.net/, Apr. 2015.

[57] QUnit – A JavaScript Unit Testing framework. http://qunitjs.com/, Apr. 2015.

[58] QuviQ A. B. http://www.quviq.com, Apr. 2015.

[59] R. http://www.mundo-r.com/, Apr. 2015.

[60] Ramada Hotels. http://www.ramada.com/, Apr. 2015.

[61] SEI – Software Engineering Institute. http://www.sei.cmu.edu/, Apr. 2015.

[62] T3DevKit. http://www.irisa.fr/tipi/wiki/doku.php/\-t3devkit, Apr. 2015.

[63] TalkEasy – Festnetz Mobilfunk Internet Digital-TV. http://www.talkeasy.ch/, Apr. 2015.

[64] Telefon, TV und Internet von NetCologne. https://www.netcologne.de/, Apr. 2015.

[65] The Apache HTTP Server Project. http://httpd.apache.org/, Apr. 2015.

[66] The Diva Hotel. http://www.thedivahotel.com/, Apr. 2015.

[67] The WebKit Open Source Project. https://www.webkit.org/, Apr. 2015.

[68] TRex – TTCN-3 Refactoring and Metrics Tool. http://www.trex.informatik.uni-goettingen.de/trac, Apr. 2015.

[69] Tsung. http://tsung.erlang-projects.org/, Apr. 2015.

[70] TTCN Lab of University of Science and Technology of China. http://ttcn.ustc.edu.cn/MainPageEn.html, Apr. 2015.

[71] UDC – Universidade da Coruña. http://www.udc.es/, Apr. 2015.

[72] UDC TV. http://www.udctv.es/, Apr. 2015.

[73] Verizon FiOS. http://www.verizon.com/home/fiostv/, Apr. 2015.

[74] Wowza – Media Streaming Server and Cloud Solutions. http://www.wowza.com/, Apr. 2015.

[75] XEmacs – The next generation of Emacs. http://www.xemacs.org/, Apr. 2015.

[76] XMLTV. http://www.xmltv.org/, Apr. 2015.

[77] XMLUnit – JUnit and NUnit testing for XML. http://xmlunit.sourceforge.net/, Apr. 2015.

[78] YouTube. http://www.youtube.com/, Apr. 2015.

[79] A. Abran, P. Bourque, R. Dupuis, and J. W. Moore, editors. *Guide to the Software Engineering Body of Knowledge - SWEBOK*. IEEE Press, 2001.

[80] Ahern, R. Turner, and A. Clouse. *CMMI Distilled: A Practical Introduction to Integrated Process Improvement*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2003.

[81] S. Ali, M. Z. Iqbal, A. Arcuri, and L. Briand. A Search-Based OCL Constraint Solver for Model-Based Test Data Generation. In *Proceedings of 11th International Conference on Quality Software (QSIC'11), Madrid, Spain, July 13-14, 2011*, pages 41–50. IEEE Computer Society, 2011.

[82] J. B. Almeida, M. J. Frade, J. S. Pinto, and S. M. d. Sousa. *Rigorous Software Development: An Introduction to Program Verification*. Undergraduate Topics in Computer Science. Springer-Verlag London, 2011.

[83] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, 2008.

[84] J. Armstrong. *Programming Erlang: Software for a Concurrent World*. Pragmatic Bookshelf, 2007.

[85] J. Armstrong, R. Virding, C. Wikström, and M. Williams. *Concurrent Programming in Erlang*. Prentice-Hall, Inc., 2nd edition, 1996.

[86] T. Arts. On Shrinking Randomly Generated Load Tests. In *Proceedings of 13th ACM SIGPLAN Workshop on Erlang (Erlang'14), Gothenburg, Sweden, September 1-3, 2014*, pages 25–31. ACM, 2014.

[87] T. Arts, L. M. Castro, and J. Hughes. Testing Erlang Data Types with Quviq QuickCheck. In *Proceedings of 7th ACM SIGPLAN workshop on ERLANG (Erlang'08), Victoria, BC, Canada, September 20-28, 2008*, pages 1–8. ACM, 2008.

[88] T. Arts, M. A. Francisco, D. Corbacho, C. B. Earle, L.-A. Fredlund, L. M. Castro, S. Thompson, J. Derrick, and other project members. D6.5 Pilots in Property Based Testing. Technical report, Prowess: Property Based Testing of Web services, 2015.

[89] T. Arts, J. Hughes, J. Johansson, and U. Wiger. Testing telecoms software with Quviq QuickCheck. In *Proceedings of 5th ACM SIGPLAN Workshop on Erlang (Erlang'06), Portland, Oregon, USA, September 18-20, 2006*, pages 2–10. ACM, 2006.

[90] T. Arts, J. Hughes, U. Norell, and H. Svensson. Testing AUTOSAR software with QuickCheck. In *Joint Proceedings of IEEE International Conference on Software Testing, Verification and Validation (ICST'15) and Testing: Academic & Industrial Conference - Practice and Research Techniques (TAIC PART'15), Graz, Austria, 17 April 2015*. IEEE, 2015.

[91] T. Arts, P. L. Seijas, and S. Thompson. Extracting quickcheck specifications from eunit test cases. In *Proceedings of 10th ACM SIGPLAN Workshop on Erlang (Erlang'11), Tokyo, Japan, September, 2011*, pages 62–71. ACM, 2011.

[92] T. Arts and S. Thompson. From Test Cases to FSMs: Augmented Test-driven Development and Property Inference. In *Proceedings of 9th ACM SIGPLAN Workshop on Erlang (Erlang'10), Baltimore, Maryland, USA, September 27-29, 2010*, pages 1–12. ACM, 2010.

[93] M. Asada and P. M. Yan. Strengthening Software Quality Assurance. *The Hewlett-Packard Journal*, 49(2):89–97, May 1998.

[94] A. Askarunisa, A. Abirami, and S. Mohan. A test case reduction method for semantic based web services. In *Proceedings of 2nd International Conference on Computing, Communication and Networking Technologies (ICCCNT'10), Karur, India, July 29-31, 2010*, pages 1–7. IEEE, 2010.

[95] D. Astels. *Test Driven Development: A Practical Guide*. Prentice Hall PTR, 2003.

[96] X. Bai, S. Lee, W. Tsai, and Y. Chen. Ontology-based test modeling and partition testing of web services. In *Proceedings of IEEE International Conference on Web Services (ICWS'08), Beijing, China, September 23-26, 2008*, pages 465–472. IEEE, 2008.

[97] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, and C. Williams. *Model-Driven Testing. Using the UML Testing Profile*. Springer-Verlag, 2008.

[98] G. Banga and P. Druschel. Measuring the Capacity of a Web Server Under Realistic Loads. *World Wide Web*, 2(1–2):69–83, Kluwer Academic Publishers, Jan. 1999.

[99] M. Barnett, K. Leino, and W. Schulte. The Spec# Programming System: An Overview. In *International Workshop on Construction and Analysis of Safe, Secure and Interoperable Smart devices, (CASSIS'04), Marseille, France, March 10-14, 2004. Revised Selected Papers*, volume 3362 of *Lecture Notes in Computer Science*, pages 49–69. Springer Berlin Heidelberg, 2004.

[100] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini. WS-TAXI: A WSDL-based testing tool for web services. In *Proceedings of 2nd International Conference on Software Testing Verification and Validation, (ICST'09), Denver, CO, April 1-4, 2009*, pages 326–335. IEEE, 2009.

[101] B. Baudry, Y. L. Sunyé, and J.-M. Jézéquel. Towards a 'Safe' Use of Design Patterns to Improve OO Software Testability. In *Proceedings of 12th International Symposium on Software Reliability Engineering (ISSRE'01), Hong Kong, China, November 27-30, 2001*, pages 324–329. IEEE, 2001.

[102] H. Baumeister, A. Knapp, and M. Wirsing. Property-Driven Development. In *Proceedings of 2nd International Conference on Software Engineering and*

Formal Methods (SEFM'04), Beijing, China, September 28-30, 2004, pages 96–102. IEEE Computer Society, 2004.

[103] BBC. Monthly Performance Pack August 2014. http://downloads.bbc.co.uk/mediacentre/iplayer/iplayer-performance-aug14.pdf, Apr. 2015.

[104] BBC. Monthly Performance Pack September 2012. http://downloads.bbc.co.uk/mediacentre/iplayer/iplayer-performance-sep12.pdf, Apr. 2015.

[105] K. Beck. Simple smalltalk testing: with patterns. Technical Report 4 (2), 1994.

[106] K. Beck. *Kent Beck's Guide to Better Smalltalk*. Cambridge University Press, 1998.

[107] K. Beck. *Test Driven Development By Example*. Addison-Wesley Longman Publishing Co., Inc., 2002.

[108] K. Beck and C. Andres. *Extreme Programming Explained: Embrace Change*. Addison Wesley Professional, 2nd edition, 2004.

[109] K. Beck and M. Fowler. *Planning Extreme Programming*. Addison Wesley, 2000.

[110] B. Beizer. *Software System Testing and Quality Assurance*. Van Nostrand Reinhold Co., 1984.

[111] B. Beizer. *Software Testing Techniques*. Van Nostrand Reinhold Co., 2nd edition, 1990.

[112] B. Beizer. *Black-box Testing: Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, Inc., 1995.

[113] C. Benac Earle, L.-A. Fredlund, A. Herranz, and J. Mariño. Jsongen: A Quickcheck Based Library for Testing JSON Web Services. In *Proceedings of 13th ACM SIGPLAN Workshop on Erlang (Erlang'14), Gothenburg, Sweden, September 1-3, 2014*, pages 33–41. ACM, 2014.

[114] M. Benattou, J. M. Bruel, and N. Hameurlain. Generating Test Data from OCL Specification. In *Proceedings of European Conference on Object-Oriented Programming (ECOOP'02). Workshop on Integration and Transformation of UML models, University of Málaga, Spain, June 10-14, 2002*.

[115] G. Bernot, M. C. Gaudel, and B. Marre. Software Testing Based on Formal Specifications: A Theory and a Tool. *Software Engineering Journal*, 6(6):387–405, IET, Nov. 1991.

[116] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli. Automatic synthesis of behavior protocols for composable web-services. In *Proceedings of 7th joint meeting of the European software engineering conference and the ACM SIGSOFT symposium on The foundations of software engineering, (ESEC/FSE'09), Amsterdam, The Netherlands, August 24-28, 2009*, pages 141–150. ACM, 2009.

[117] R. Black. *Managing the Testing Process: Practical Tools and Techniques for Managing Hardware and Software Testing*. Wiley Publishing, 3rd edition, 2009.

[118] B. Blanchard and W. Fabrycky. *Systems Engineering and Analysis*. Prentice-Hall international series in industrial and systems engineering. Pearson Prentice Hall, 2006.

[119] S. Blom, N. Ioustinova, J. van de Pol, A. Rennoch, and N. Sidorova. Simulated Time for Testing Railway Interlockings with TTCN-3. In *5th International Workshop on Formal Approaches to Software Testing (FATES'05), Edinburgh, UK, July 11, 2005. Revised Selected Papers*, volume 3997 of *Lecture Notes in Computer Science*, pages 1–15. Springer Berlin Heidelberg, 2005.

[120] B. W. Boehm. Guidelines for Verifying and Validating Software Requirements and Design Specifications. In *Proceedings of European Conference on Applied Information Technology of the International Federation for Information Processing (Euro IFIP'79), London, September 25-28, 1979*, pages 711–719. North Holland, 1979.

[121] B. W. Boehm. A Spiral Model of Software Development and Enhancement. *ACM SIGSOFT Software Engineering Notes*, 11(4):14–24, ACM, Aug. 1986.

[122] B. W. Boehm. A View of 20th and 21st Century Software Engineering. In *Proceedings of 28th International Conference on Software Engineering (ICSE'06), Shanghai, China, May 20-28, 2006*, pages 12–29. ACM, 2006.

[123] B. W. Boehm and V. R. Basili. Software Defect Reduction Top 10 List. *Computer*, 34(1):135–137, IEEE Computer Society Press, Jan. 2001.

[124] B. W. Boehm, R. Valerdi, and E. Honour. The ROI of Systems Engineering: Some Quantitative Results for Software-Intensive Systems. *Systems Engineering*, 11(3):221–234, Wiley Subscription Services, Inc., A Wiley Company, Apr. 2008.

[125] G. Booch. *Object-oriented Analysis and Design with Applications*. Benjamin-Cummings Publishing Co., Inc., 2nd edition, 1994.

[126] G. Booch, J. Rumbaugh, and I. Jacobson. *The Unified Modeling Language User Guide*. Addison-Wesley Professional, 2nd edition, 2005.

[127] F. Brooks. *The Mythical Man-month: Essays on Software Engineering*. Addison-Wesley, 1995.

[128] A. D. Brucker, M. P. Krieger, D. Longuet, and B. Wolff. A specification-based test case generation method for UML/OCL. In *Workshop on OCL and Textual Modelling (MODELS'10), Oslo, Norway, October 2-8, 2010. Reports and Revised Selected Papers*, volume 6627 of *Lecture Notes in Computer Science*, pages 334–348. Springer Berlin Heidelberg, 2011.

[129] J. Cabot and M. Gogolla. Object Constraint Language (OCL): A Definitive Guide. In *12th International School on Formal Methods for the Design of Computer, Communication, and Software Systems (SFM'12), Bertinoro, Italy, June 18-23, 2012. Advanced Lectures*, volume 7320 of *Lecture Notes in Computer Science*, pages 58–90. Springer Berlin Heidelberg, 2012.

[130] C. Cadar and K. Sen. Symbolic Execution for Software Testing: Three Decades Later. *Communications of the ACM*, 56(2):82–90, ACM, Feb. 2013.

[131] B. Calder, J. Courtney, B. Foote, L. Kyrnitszke, D. Rivas, C. Saito, J. V. Loo, and T. Ye. *Java TV API Technical Overview: The Java TV API Whitepaper*. Sun microsystems, 2000.

[132] D. N. Card, F. E. McGarry, and G. T. Page. Evaluating Software Engineering Technologies. *IEEE Transactions on Software Engineering*, 13(7):845–851, IEEE Press, July 1987.

[133] R. Carlsson and M. Rémond. EUnit: A Lightweight Unit Testing Framework for Erlang. In *Proceedings of 5th ACM SIGPLAN Workshop on Erlang (Erlang'06), Portland, Oregon, USA, September 18-20, 2006*, pages 1–1. ACM, 2006.

[134] R. H. Carver and K.-C. Tai. Replay and Testing for Concurrent Programs. *IEEE Software*, 8(2):66–74, IEEE Computer Society Press, Mar. 1991.

[135] L. M. Castro and T. Arts. Testing Data Consistency of Data-Intensive Applications Using QuickCheck. In *Proceedings of 10th Spanish Conference on Programming and Languages (PROLE'10), Valencia, Spain, September 8–10, 2010. Revised Selected Papers*, volume 271 of *Electronic Notes in Theoretical Computer Science*, pages 41–62. Elsevier Science Publishers, 2011.

[136] L. M. Castro and M. A. Francisco. A Language-independent Approach to Black-box Testing Using Erlang As Test Specification Language. *Journal of Systems and Software*, 86(12):3109–3122, Elsevier Science Inc., Dec. 2013.

[137] L. M. Castro, M. A. Francisco, and V. M. Gulías. Applications integration: a testing experience. In *Proceedings of 6th Workshop on System Testing and Validation (STV'08), Madrid, Spain, December 13, 2008.*

[138] L. M. Castro, M. A. Francisco, and V. M. Gulías. A Practical Methodology for Integration Testing. In *12th International Conference on Computer Aided Systems Theory (EUROCAST'09), Las Palmas de Gran Canaria, Spain, February 15-20, 2009. Revised Selected Papers*, volume 5717 of *Lecture Notes in Computer Science*, pages 881–888. Springer Berlin Heidelberg, 2009.

[139] L. M. Castro, M. A. Francisco, and V. M. Gulías. Testing Integration of Applications with QuickCheck. In *Extended Abstracts of Twelve International Conference on Computer Aided Systems Theory (EUROCAST'09), Las Palmas de Gran Canaria, Spain, February 15-20, 2009*, pages 299–300. IUCTC Universidad de Las Palmas de Gran Canaria, 2009.

[140] Y. Cheon and C. Avila. Automating Java Program Testing Using OCL and AspectJ. In *Proceedings of 7th International Conference on Information Technology: New Generations (ITNG'10), Las Vegas, Nevada, USA, April 12-14, 2010*, pages 1020–1025. IEEE Computer Society, 2010.

[141] Y. Cheon and G. T. Leavens. A Simple and Practical Approach to Unit Testing: The JML and JUnit Way. In *Proceedings of 16th European Conference on Object-Oriented Programming (ECOOP'02), Málaga, Spain, June 10–14, 2002*, volume 2374 of *Lecture Notes in Computer Science*, pages 231–255. Springer Berlin Heidelberg, 2002.

[142] J. J. Chilenski and S. P. Miller. Applicability of modified condition/decision coverage to software testing. *Software Engineering Journal*, 9(5):193–200, IET, Sept. 1994.

[143] M. B. Chrissis, M. Konrad, and S. Shrum. *CMMI: Guidelines for Process Integration and Product Improvement*. Addison-Wesley, 2nd edition, 2007.

[144] K. Claessen and J. Hughes. QuickCheck: a lightweight tool for random testing of Haskell programs. In *Proceedings of 5th ACM SIGPLAN International Conference on Functional programming (ICFP'00), Montreal, Canada, September 18-21, 2000*, pages 268–279. ACM, 2000.

[145] E. M. Clarke and J. M. Wing. Formal Methods: State of the Art and Future Directions. *ACM Computing Surveys*, 28(4):626–643, ACM, Dec. 1996.

[146] P. Coad, J. d. Luca, and E. Lefebvre. *Java Modeling Color with Uml: Enterprise Components and Process*. Prentice Hall PTR, 1999.

[147] P. Coad and E. Yourdon. *Object-oriented Analysis*. Yourdon Press, 2nd edition, 1991.

[148] A. Cockburn. *Crystal Clear a Human-powered Methodology for Small Teams*. Addison-Wesley Professional, 2004.

[149] A. Coen-Porisini, G. Denaro, C. Ghezzi, and M. Pezzé. Using Symbolic

Execution for Verifying Safety-critical Systems. In *Proceedings of 8th European Software Engineering Conference Held Jointly with 9th ACM SIGSOFT International Symposium on Foundations of Software Engineering (ESEC/FSE'01), Vienna, Austria, September 10-14, 2001*, pages 142–151. ACM, 2001.

[150] D. Cohen, M. Lindvall, and P. Costa. An Introduction to Agile Methods. *Advances in Computers*, 62:1–66, Elsevier, 2004.

[151] D. Corbacho. Tutorial: Load Testing Made Easy. In *Erlang User Conference, Stockholm, June 11, 2014*.

[152] R. D. Craig and S. P. Jaskiel. *Systematic Software Testing*. Artech House, Inc., 2002.

[153] D. Crockford. *The application/json Media Type for JavaScript Object Notation (JSON)*. IETF, July 2006. RFC 4627.

[154] O. J. Dahl, E. W. Dijkstra, and C. A. R. Hoare. *Structured Programming*. Academic Press Ltd., 1972.

[155] E. Daka and G. Fraser. A Survey on Unit Testing Practices and Problems. In *Proceedings of 25th International Symposium on Software Reliability Engineering (ISSRE'14), Naples, Italy, November 3-6, 2014*, pages 201–211. IEEE Computer Society, 2014.

[156] A. M. Davis. *Software Requirements: Objects, Functions, and States*. Prentice-Hall, Inc., 1993.

[157] J. Day and H. Zimmermann. The OSI reference model. *Proceedings of the IEEE*, 71(12):1334–1340, 1983.

[158] T. DeMarco. Classics in Software Engineering. chapter Structured Analysis and System Specification, pages 409–424. Yourdon Press, 1979.

[159] B. Demuth. The Dresden OCL Toolkit and its Role in Information Systems Development. In *Proceedings of 13th International Conference on Information Systems Development: Methods and Tools, Theory and Practice Conference, Advances in Theory, Practice and Education (ISD'04), Vilnius Gediminas Technical University, Lithuania. September 9-11, 2004*.

[160] B. Demuth, S. Loecher, and S. Zschaler. Structure of the Dresden OCL Toolkit. In *Proceedings of 2nd International Fujaba Days "MDA with UML and Rule-based Object Manipulation", Technical University of Darmstadt, Germany, September 15-17, 2004*, pages 1–2.

[161] J. B. Dennis. Modularity. In *Software Engineering, An Advanced Course, Reprint of the First Edition*, volume 30 of *Lecture Notes in Computer Science*, pages 128–182. Springer Berlin Heidelberg, 1975.

[162] J. Derrick, N. Walkinshaw, T. Arts, C. B. Earle, F. Cesarini, L.-A. Fredlund, V. Gulías, J. Hughes, and S. Thompson. Property-Based Testing – The ProTest Project. In *8th International Symposium in Formal Methods for Components and Objects (FMCO'09), Eindhoven, The Netherlands, November 4-6, 2009. Revised Selected Papers*, volume 6286 of *Lecture Notes in Computer Science*, pages 250–271. Springer Berlin Heidelberg, 2010.

[163] G. A. Di Lucca and A. R. Fasolino. Testing Web-based Applications: The State of the Art and Future Trends. *Information and Software Technology*, 48(12):1172–1186, Butterworth-Heinemann, Dec. 2006.

[164] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of 1st ACM International Workshop on Empirical Assessment of Software Engineering Languages and Technologies (WEASELTech'07), Atlanta, Georgia, November 5-9, 2007*, pages 31–36. ACM, 2007.

[165] A. C. Dias Neto, R. Subramanyan, M. Vieira, and G. H. Travassos. A survey on model-based testing approaches: a systematic review. In *Proceedings of 1st ACM international Workshop on Empirical assessment of Software Engineering Languages and Technologies (WEASELTech'07), Atlanta, GA, USA, November 5-9, 2007*, pages 31–36. ACM, 2007.

[166] J. Dick and A. Faivre. Automating the Generation and Sequencing of Test Cases from Model-Based Specifications. In *Proceedings of 1st International Symposium of Formal Methods Europe on Industrial-Strength Formal Methods (FME'93), Odense, Denmark, April 19–23, 1993*, volume 670 of *Lecture Notes in Computer Science*, pages 268–284. Springer Berlin Heidelberg, 1993.

[167] E. W. Dijkstra. The Humble Programmer. *Communications of the ACM*, 15(10):859–866, ACM, Oct. 1972.

[168] D. Draheim, J. Grundy, J. Hosking, C. Lutteroth, and G. Weber. In *Proceedings of 10th European Conference on Software Maintenance and Reengineering (CSMR'06), Universita di Bari - Bari, Italy, March 22-24, 2006*.

[169] R. G. Dromey. From Requirements to Design: Formalizing the Key Steps. In *Proceedings of 1st International Conference on Software Engineering and Formal Methods (SEFM'03), Brisbane, Queensland, Australia, September 22-27, 2003*, pages 2–11. IEEE Computer Society, 2003.

[170] J. W. Duran and S. Ntafos. An Evaluation of Random Testing. *IEEE Transactions on Software Engineering*, SE-10(4):438–444, IEEE, July 1984.

[171] A. Egner, F. Moldoveanu, A. Moldoveanu, V. Asavei, A. Morar, and C. Boiangiu. Testing the interoperability of HL7-based applications using

TTCN-3. In *Annals of DAAAM for 2010 & Proceedings of 21st International DAAAM Symposium, Zadar, Croatia, October 20-23, 2010*, pages 1279–1280. DAAAM International Vienna, 2010.

[172] N. S. Eickelmann and D. J. Richardson. What Makes One Software Architecture More Testable Than Another? In *Joint Proceedings of Second International Software Architecture Workshop (ISAW'96) and International Workshop on Multiple Perspectives in Software Development (Viewpoints'96) on SIGSOFT'96 Workshops (ISAW'96), San Francisco, California, USA, 1996*, pages 65–67. ACM, 1996.

[173] K. E. Emam and A. G. Koru. A Replicated Survey of IT Software Project Failures. *IEEE Software*, 25(5):84–90, IEEE Computer Society Press, Aug. 2008.

[174] T. Erl. *Service-Oriented Architecture: Concepts, Technology, and Design*. Prentice Hall PTR, 2005.

[175] European Telecommunications Standards Institute. *ETSI ES 201 873-2: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 2: TTCN-3 Tabular presentation Format (TFT), V3.2.1*, Feb. 2007.

[176] European Telecommunications Standards Institute. *ETSI ES 201 873-3: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 3: TTCN-3 Graphical presentation Format (GFT), V3.2.1*, Feb. 2007.

[177] European Telecommunications Standards Institute. *Digital Video Broadcasting (DVB); Multimedia Home Platform (MHP) Specification 1.1.3*, 2012. Technical Specification.

[178] European Telecommunications Standards Institute. *ETSI ES 201 873-1: Methods for Testing and Specification (MTS); The Testing and Test Control Notation version 3; Part 1: TTCN-3 Core Language, V4.6.1*, June 2014.

[179] J. Evain. The Multimedia Home Platform – an overview. Technical Report 275, EBU Technical Department, May 1998.

[180] P. Farrell-Vinay. *Manage software testing*. Auerbach Publishers, 2008.

[181] R. T. Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[182] G. Fink and M. Bishop. Property-based testing: a new approach to testing for assurance. *ACM SIGSOFT Software Engineering Notes*, 22(4):74–80, ACM, July 1997.

[183] M. Forsberg and A. Ranta. Labelled BNF: a highlevel formalism for defining well-behaved programming languages. In *14th Nordic Workshop on Programming Theory (NWPT'02), Tallinn, Estonia, November 20-22, 2002*, volume 52 of *Proceedings of the Estonian Academy of Sciences: Physics and Mathematics*, pages 356—377, 2003.

[184] M. Fowler. *Refactoring: Improving the Design of Existing Code*. Addison Wesley, 1999.

[185] M. A. Francisco. Análisis, Diseño e Implementación de un Sistema de Televisión a la Carta para Entornos Hosteleros. Master's thesis, University of A Coruña, Dec. 2005.

[186] M. A. Francisco and L. M. Castro. Uso de propiedades y modelos para las pruebas de sistemas distribuidos basados en la integración de componentes heterogéneos. In *Proceedings of 13th Spanish Conference on Programming and Languages (PROLE'13) and V Workshop on Functional Programming (TPF), Madrid, Spain, September 17-20, 2013*, pages 37–39.

[187] M. A. Francisco and L. M. Castro. Automatic Generation of Test Models and Properties from UML Models with OCL Constraints. In *Proceedings of 12th Workshop on OCL and Textual Modelling (OCL'12, co-located with MODELS'12), Innsbruck, Austria, October 1-5, 2012*, pages 49–54. ACM, 2012.

[188] M. A. Francisco, L. M. Castro, and D. Corbacho. A Property-based Load Testing Approach: a case study. *Journal of Systems and Software (submitted)*, 2015.

[189] M. A. Francisco, L. M. Castro, and V. M. Gulías. Uso de propiedades abstractas para especificación de pruebas funcionales de caja negra. In *Proceedings of 11th Spanish Conference on Programming and Languages (PROLE'11), A Coruña, Spain, September 5-7, 2011*, pages 81–95.

[190] M. A. Francisco, M. López, H. Ferreiro, and L. M. Castro. Turning Web Services Descriptions into Quickcheck Models for Automatic Testing. In *Proceedings of 12th ACM SIGPLAN Workshop on Erlang (Erlang'13), Boston, Massachusetts, USA, September 28, 2013*, pages 79–86. ACM, 2013.

[191] L.-A. Fredlund, C. Benac Earle, A. Herranz, and J. Marino. Property-Based Testing of JSON Based Web Services. In *Proceedings of IEEE International Conference on Web Services (ICWS'14), Anchorage, AK, USA, June 27 - July 2, 2014*, pages 704–707. IEEE, 2014.

[192] L.-A. Fredlund and H. Svensson. McErlang: A Model Checker for a Distributed Functional Programming Language. In *Proceedings of 12th ACM*

SIGPLAN International Conference on Functional Programming (ICFP'07), Freiburg, Germany, October 1-3, 2007, pages 125–136. ACM, 2007.

[193] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-oriented Software*. Addison-Wesley Longman Publishing Co., Inc., 1995.

[194] D. Gelperin and B. Hetzel. The Growth of Software Testing. *Communications of the ACM*, 31(6):687–695, ACM, June 1988.

[195] D. Georgakopoulos and M. P. Papazoglou. *Service-Oriented Computing*. The MIT Press, 2008.

[196] J. A. Goguen and C. Linde. Techniques for Requirements Elicitation. In *Proceedings of IEEE International Symposium on Requirements Engineering (RE'93), San Diego, CA, USA, January 4-6, 1993*, pages 152–164. IEEE Computer Society, 1993.

[197] J. Grabowski. TTCN-3 – A new Test Specification Language for Black-Box Testing of Distributed Systems. In *Proceedings of 17th International Conference and Exposition on Testing Computer Software (TCS'00), Theme: Testing Technology vs.Testers' Requirements, Washington D.C., June, 2000.*

[198] M. Grossman, J. E. Aronson, and R. V. McCarthy. Does UML Make the Grade? Insights from the Software Development Community. *Information and Software Technology*, 47(6):383–397, Butterworth-Heinemann, Apr. 2005.

[199] J. Grossmann, D. A. Serbanescu, and I. Schieferdecker. Testing Embedded Real Time Systems with TTCN-3. In *Proceedings of 2nd International Conference on Software Testing Verification and Validation, (ICST'09), Denver, CO, April 1-4, 2009*, pages 81–90. IEEE Computer Society, 2009.

[200] V. M. Gulías, M. Barreiro, and J. L. Freire. VoDKA: Developing a Video-on-Demand Server using Distributed Functional Programming. *Journal on Functional Programming*, 15(3):403–430, Cambridge University Press, May 2005.

[201] J. Guo, Y. Liao, and B. Parviz. A Survey of J2EE Application Performance Management Systems. In *Proceedings of IEEE International Conference on Web Services (ICWS'04), San Diego, CA, USA, July 6-9, 2004*, pages 724–731. IEEE Computer Society, 2004.

[202] P. Hamill. *Unit Test Frameworks*. O'Reilly, 1st edition, 2004.

[203] D. Hamlet. When Only Random Testing Will Do. In *Proceedings of 1st International Workshop on Random Testing (RT'06), Portland, ME, USA, July 17-20, 2006*, pages 1–9. ACM, 2006.

[204] D. Hatley, P. Hruschka, and I. A. Pirbhai. *Process for System Architecture and Requirements Engineering*. Dorset House Publishing Co., Inc., 2000.

[205] R. Heckel and M. Lohmann. Towards Contract-based Testing of Web Services. In *Proceedings of International Workshop on Test and Analysis of Component Based Systems (TACoS'04), Barcelona, Spain, March 27–28, 2004*, volume 116 of *Electronic Notes in Theoretical Computer Science*, pages 145–156. Elsevier Science Publishers B. V., 2005.

[206] A. Hedayat, N. Sloane, and J. Stufken. *Orthogonal Arrays: Theory and Applications*. Springer Series in Statistics. Springer-Verlag New York, 1999.

[207] B. Hetzel. *The Complete Guide to Software Testing*. QED Information Sciences, Inc., 2nd edition, 1988.

[208] R. M. Hierons, K. Bogdanov, J. P. Bowen, R. Cleaveland, J. Derrick, J. Dick, M. Gheorghe, M. Harman, K. Kapoor, P. Krause, G. Lüttgen, A. J. H. Simons, S. Vilkomir, M. R. Woodward, and H. Zedan. Using Formal Specifications to Support Testing. *ACM Computing Surveys*, 41(2):9:1–9:76, ACM, Feb. 2009.

[209] J. A. Highsmith, III. *Adaptive Software Development: A Collaborative Approach to Managing Complex Systems*. Dorset House Publishing Co., Inc., 2000.

[210] S. Hildreth. Automating the 40 Monkeys. Technical report, Computerworld, 2004.

[211] J. Hughes. QuickCheck Testing for Fun and Profit. In *Proceedings of 9th International Symposium on Practical Aspects of Declarative Languages (PADL'07), Nice, France, January 14-15, 2007*, volume 4354 of *Lecture Notes in Computer Science*, pages 1–32. Springer Berlin Heidelberg, 2007.

[212] A. Hunt and D. Thomas. *Pragmatic Unit Testing in Java with JUnit*. The Pragmatic Programmers, 2003.

[213] A. Hunt and D. Thomas. *Pragmatic Unit Testing in C# with Nunit*. The Pragmatic Programmers, 2nd edition, 2007.

[214] J. Hutchinson, J. Whittle, M. Rouncefield, and S. Kristoffersen. Empirical Assessment of MDE in Industry. In *Proceedings of 33rd International Conference on Software Engineering (ICSE'11), Waikiki, Honolulu, HI, USA, May 21-28, 2011*, pages 471–480. ACM, 2011.

[215] IBM Systems Sciences Institute. *Implementing Software Inspections*, 1981.

[216] I. Jacobson. *Object-oriented Software Engineering: A Use Case Driven Approach*. Addison Wesley (ACM Press), 1992.

[217] I. Jacobson, G. Booch, and J. Rumbaugh. *The Unified Software Development Process*. Addison-Wesley Longman Publishing Co., Inc., 1999.

[218] D. Janzen and H. Saiedian. Test-Driven Development: Concepts, Taxonomy, and Future Direction. *Computer*, 38(9):43–50, IEEE Computer Society Press, Sept. 2005.

[219] C. D. H. Jeffery E. Payne, Roger T. Alexander. Design-for-Testability for Object-Oriented Software. *Object Magazine*, 7(5):34–43, May 1997.

[220] Z. Jin and A. J. Offutt. Coupling-based criteria for integration testing. *The Journal of Software Testing, Verification and Reliability*, 8(3):133–154, John Wiley & Sons, Ltd., Sept. 1998.

[221] C. Jones. Social and technical reasons for software project failures. *CrossTalk: The Journal of Defense Software Engineering*, 19(6):4–9, June 2006.

[222] P. C. Jorgensen. *Software Testing: A Craftsman's Approach*. Auerbach Publications, 3rd edition, 2008.

[223] JSON-RPC Working Group. JSON-RPC 2.0 Specification. `http://www.jsonrpc.org/`, Jan. 2013.

[224] T. Jung. Java implementation of QuickChek. `https://bitbucket.org/blob79/quickcheck`, Apr. 2015.

[225] C. Kaner, J. L. Falk, and H. Q. Nguyen. *Testing Computer Software*. John Wiley & Sons, Inc., 2nd edition, 1999.

[226] M. Karaorman, U. Hölzle, and J. L. Bruno. jContractor: A Reflective Java Library to Support Design by Contract. In *Proceedings of 2nd International Conference on Meta-Level Architectures and Reflection (Reflection'99), Saint-Malo, France, July 19–21, 1999*, pages 175–196. Springer Berlin Heidelberg, 1999.

[227] K. Karhu, T. Repo, O. Taipale, and K. Smolander. Empirical Observations on Software Testing Automation. In *Proceedings of 2nd International Conference on Software Testing Verification and Validation, (ICST'09), Denver, CO, April 1-4, 2009*, pages 201–209. IEEE Computer Society, 2009.

[228] M. E. Khan and F. Khan. A Comparative Study of White Box, Black Box and Grey Box Testing Techniques. *International Journal of Advanced Computer Science and Applications*, 3(6):12–15, 2012.

[229] F. Khomh and Y.-G. Gueheneuce. Do Design Patterns Impact Software Quality Positively? In *Proceedings of 12th European Conference on Software Maintenance and Reengineering (CSMR'08), Athens, Greece, April 1-4, 2008*, pages 274–278. IEEE Computer Society, 2008.

[230] J. C. King. Symbolic Execution and Program Testing. *Communications of the ACM*, 19(7):385–394, ACM, July 1976.

[231] R. Kramer. iContract - The Java(tm) Design by Contract(tm) Tool. In *Proceedings of 6th International Conference on Technology of Object-Oriented Languages and Systems (TOOLS'98), Santa Barbara, CA, USA, August 3-7, 1998*, pages 295–307. IEEE Computer Society, 1998.

[232] D. C. Kung, J. Gao, and C.-H. Kung. *Testing Object-Oriented Software*. Wiley-IEEE Computer Society Press, 1998.

[233] P. Lamela Seijas, S. Thompson, and M. A. Francisco. Model extraction and test generation from JUnit suites. In *University of Kent School of Computing Conference (UKSCC'15), Kent, United Kingdom, June 15, 2015*.

[234] P. Lamela Seijas, S. Thompson, and M. A. Francisco. Model extraction and test generation from JUnit suites. In *7th International Symposium on Search-Based Software Engineering (SSBSE'15), Bergamo, Italy, September 5-7, 2015 (submitted)*.

[235] L. Lampropoulos and K. F. Sagonas. Automatic WSDL-guided Test Case Generation for PropEr Testing of Web Services. In *Proceedings of 8th International Workshop on Automated Specification and Verification of Web Systems (WWV'12), Stockholm, June 16, 2012*, volume 98 of *Electronic Proceedings in Theoretical Computer Science*, pages 3–16, 2012.

[236] P. A. Laplante and C. J. Neill. "The Demise of the Waterfall Model Is Imminent" and Other Urban Myths. *Queue - Game Development*, 1(10):10–15, ACM, Feb. 2004.

[237] C. Larman and V. R. Basili. Iterative and Incremental Development: A Brief History. *Computer*, 36(6):47–56, IEEE Computer Society Press, June 2003.

[238] S. S. Laurent, E. Dumbill, and J. Johnston. *Programming Web Services with XML-RPC*. O'Reilly & Associates, Inc., 2001.

[239] C. League. QCheck/SML. `http://contrapunctus.net/league/haques/qcheck/`, Apr. 2015.

[240] G. T. Leavens and Y. Cheon. *Design by Contract with JML*. www.jmlspecs.org, 2006.

[241] J. Lee, S. Kang, and D. Lee. Survey on software testing practices. *IET Software*, 6(3):275–282, IET, June 2012.

[242] H. Li and S. Thompson. Tool Support for Refactoring Functional Programs. In *Proceedings of ACM SIGPLAN Symposium on Partial Evaluation and Semantics-based Program Manipulation (PEPM'08), San Francisco, California, USA, January 7-8, 2008*, pages 199–203. ACM, 2008.

[243] H. Li and S. Thompson. A User-extensible Refactoring Tool for Erlang Programs. Technical report, School of Computing, Univ. of Kent, UK, 2011.

[244] H. Li and S. Thompson. A Domain-specific Language for Scripting Refactorings in Erlang. In *Proceedings of 15th International Conference on Fundamental Approaches to Software Engineering (FASE'12), Tallinn, Estonia, March 24 - April 1, 2012*, volume 7212 of *Lecture Notes in Computer Science*, pages 501–515. Springer Berlin Heidelberg, 2012.

[245] H. Li and S. Thompson. Automated API Migration in a User-extensible Refactoring Tool for Erlang Programs. In *Proceedings of 27th IEEE/ACM International Conference on Automated Software Engineering (ASE'12), Essen, Germany, September 3-7, 2012*, pages 294–297. IEEE/ACM, 2012.

[246] H. Li, S. Thompson, and T. Arts. Extracting Properties from Test Cases by Refactoring. In *4th IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW'11), Berlin, Germany, March 21-25, 2011*, pages 472–473. IEEE Computer Society.

[247] H. Li, S. Thompson, P. Lamela Seijas, and M. A. Francisco. Automating Property-based Testing of Evolving Web Services. In *Proceedings of ACM SIGPLAN Workshop on Partial Evaluation and Program Manipulation (PEPM'14), San Diego, California, USA, January 20-21, 2014*, pages 169–180. ACM, 2014.

[248] K. Li and M. Wu. *Effective Software Test Automation: Developing an Automated Software Testing Tool*. SYBEX Inc., 2004.

[249] M. Lopez, H. Ferreiro, L. M. Castro, and T. Art. A DSL for Web Services Automatic Test Data Generation. In *Proceedings of 25th International Symposium on Implementation and Application of Functional Languages (IFL'13), Nijmegen, Netherlands, August 28-30, 2013*. ACM, 2013.

[250] M. López, H. Ferreiro, M. A. Francisco, and L. M. Castro. Automatización de Pruebas para Servicios Web: Generación de Propiedades y Modelos. In *Proceedings of 13th Spanish Conference on Programming and Languages (PROLE'13) and V Workshop on Functional Programming (TPF), Madrid, Spain, September 17-20, 2013*, pages 61–75.

[251] Z. Lukasik and W. Nowakowski. Application of TTCN-3 for Testing of Railway Interlocking Systems. In *10th Conference on Transport Systems Telematics (TST'10), Katowice, Ustron, Poland, October 20-23, 2010. Selected Papers*, volume 104 of *Communications in Computer and Information Science*, pages 447–454. Springer Berlin Heidelberg, 2010.

[252] M. López and L. M. Castro. Validación de tiempos de respuesta usando pruebas basadas en propiedades. In *Proceedings of 14th Spanish Conference*

on Programming and Languages (PROLE'14), Cádiz, Spain, September 16-19, 2014, pages 117–131.

[253] M. López, H. Ferreiro, M. A. Francisco, and L. M. Castro. Automatic Generation of Test Models for Web Services Using WSDL and OCL. In *Proceedings of 11th International Conference on Service Oriented Computing (ICSOC'13), Berlin, Germany, December 2-5, 2013*, volume 8274 of *Lecture Notes in Computer Science*, pages 483–490. Springer Berlin Heidelberg, 2013.

[254] T. Mackinnon, S. Freeman, and P. Craig. Endo-testing: unit testing with mock objects. In *Extreme Programming Examined*, pages 287–301. Addison-Wesley Longman Publishing, 2001.

[255] P. Madsen. Testing by Contract - Combining Unit Testing and Design by Contract. In *Proceedings of Nordic workshop on Software Development Tools and Techniques (NWPER'02), Copenhagen, August 18-20, 2002*.

[256] E. A. Manolis Papadakis and K. Sagonas. PropEr: A QuickCheck-inspired property-based testing tool for Erlang. `http://proper.softlab.ntua.gr/`, Apr. 2015.

[257] M. V. Mantyla and C. Lassenius. What Types of Defects Are Really Discovered in Code Reviews? *IEEE Transactions on Software Engineering*, 35(3):430–448, IEEE Press, May 2009.

[258] J. Martin. *Rapid Application Development*. Macmillan Publishing Co., Inc., 1991.

[259] V. Massol and T. Husted. *JUnit in Action*. Manning Publications Co., 2003.

[260] W. M. McKeeman. Differential Testing for Software. *Digital Technical Journal*, 10(1):100–107, Digital Equipment Corporation, 1998.

[261] H. Mei and L. Zhang. A Framework for Testing Web Services and Its Supporting Tool. In *Proceedings of IEEE International Workshop on Service-Oriented System Engineering (SOSE'05), Beijing, China, October 20-21, 2005*, pages 207–214. IEEE Computer Society, 2005.

[262] J. Meier, C. Farre, P. Bansode, S. Barber, and D. Rea. *Performance Testing Guidance for Web Applications: Patterns & Practices*. Microsoft Press, 2007.

[263] D. A. Menascé. Load Testing of Web Sites. *IEEE Internet Computing*, 6(4):70–74, IEEE Educational Activities Department, July 2002.

[264] G. Meszaros. *XUnit Test Patterns: Refactoring Test Code*. Prentice Hall, 2006.

[265] B. Meyer. *Object-Oriented Software Construction*. Prentice-Hall, Inc., 1988.

[266] B. P. Miller, L. Fredriksen, and B. So. An Empirical Study of the Reliability of UNIX Utilities. *Communications of the ACM*, 33(12):32–44, ACM, Dec. 1990.

[267] R. Moiseev, S. Hayashi, and M. Saeki. Generating Assertion Code from OCL: A Transformational Approach Based on Similarities of Implementation Languages. In *Proceedings of 12th International Conference on Model Driven Engineering Languages and Systems (MODELS'09), Denver, CO, October 4-9, 2009*, volume 5795 of *Lecture Notes in Computer Science*, pages 650–664. Springer Berlin Heidelberg, 2009.

[268] I. Molyneaux. *The Art of Application Performance Testing: Help for Programmers and Quality Assurance*. O'Reilly Media, Inc., 2009.

[269] S. Morris. *Interactive TV standards: A Guide to MHP, OCAP, and JavaTV*. Elsevier, 2005.

[270] J. D. Musa. Operational Profiles in Software-Reliability Engineering. *IEEE Software*, 10(2):14–32, IEEE Computer Society Press, Mar. 1993.

[271] J. D. Musa. *Software Reliability Engineering: More Reliable Software Faster and Cheaper*. Authorhouse, 2nd edition, 2004.

[272] J. D. Musa, G. Fuoco, N. Irving, D. Kropfl, and B. Juhlin. Handbook of Software Reliability Engineering. chapter The Operational Profile, pages 167–216. McGraw-Hill, Inc., 1996.

[273] M. H. Mustafa Bozkurt and Y. Hassoun. Testing Web Services: A Survey. Technical Report TR-10-01, Department of Computer Science, King's College London, Jan. 2010.

[274] G. Myers, C. Sandler, and T. Badgett. *The Art of Software Testing*. Wiley, 3rd edition, 2011.

[275] T. Mytkowicz, A. Diwan, M. Hauswirth, and P. F. Sweeney. Evaluating the Accuracy of Java Profilers. In *Proceedings of ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'10), Toronto, Ontario, Canada, June 5-10, 2010*, pages 187–197. ACM, 2010.

[276] C. J. Neill and P. A. Laplante. Requirements Engineering: The State of the Practice. *IEEE Software*, 20(6):40–45, IEEE Computer Society Press, Nov. 2003.

[277] A. Nilsson, L. M. Castro, S. Rivas, and T. Arts. Assessing the effects of introducing a new software development process: a methodological description. *International Journal on Software Tools for Technology Transfer*, 17(1):1–17, Springer-Verlag, Apr. 2013.

[278] S. Noikajana and T. Suwannasart. An approach for web service test case generation based on web service semantics. In *Proceedings of International Conference on Semantic Web & Web Services, (SWWS'08), Las Vegas, Nevada, USA, July 14-17, 2008*, pages 171–177. CSREA press, 2008.

[279] S. Noikajana and T. Suwannasart. An improved test case generation method for web service testing from WSDL-S and OCL with pair-wise testing technique. In *Proceedings of 33rd Annual IEEE International Computer Software and Applications Conference (COMPSAC'09), Seattle, WA, USA, July 20-24, 2009*, volume 1, pages 115–123. IEEE, 2009.

[280] Object Management Group. *UML Testing Profile (UTP), Version 1.2*, Apr. 2013.

[281] Object Management Group. *Object Constraint Language, Version 2.4*, Feb. 2014.

[282] A. J. Offutt, M. J. Harrold, and P. Kolte. A Software Metric System for Module Coupling. *Journal of Systems and Software*, 20(3):295–308, Elsevier Science Inc., Mar. 1993.

[283] J. Olsson. STEVE - Performance Testing a $1 Billion SOA. In *Erlang User Conference, Stockholm, June 9-10, 2014*.

[284] R. Osherove. *The Art of Unit Testing: With Examples in .Net*. Manning Publications Co., 1st edition, 2009.

[285] T. J. Ostrand and M. J. Balcer. The Category-partition Method for Specifying and Generating Fuctional Tests. *Communications of the ACM*, 31(6):676–686, ACM, June 1988.

[286] M. Page-Jones. *The Practical Guide to Structured Systems Design*. Yourdon Press, 2nd edition, 1988.

[287] N. Paladi and T. Arts. Model Based Testing of Data Constraints: Testing the Business Logic of a Mnesia Application with Quviq QuickCheck. In *Proceedings of 8th ACM SIGPLAN workshop on ERLANG (Erlang'09), Edinburgh, Scotland, September 5, 2009*, pages 71–82. ACM, 2009.

[288] S. R. Palmer and M. Felsing. *A Practical Guide to Feature-Driven Development*. Pearson Education, 2001.

[289] M. Papadakis and K. Sagonas. A PropEr Integration of Types and Function Specifications with Property-Based Testing. In *Proceedings of 10th ACM SIGPLAN Workshop on Erlang (Erlang'11), Tokyo, Japan, September, 2011*, pages 39–50. ACM, 2011.

[290] M. P. Papazoglou and W.-J. Heuvel. Service Oriented Architectures: Approaches, Technologies and Research Issues. *The VLDB Journal - The Inter-*

national Journal on Very Large Data Bases, 16(3):389–415, Springer-Verlag New York, Inc., July 2007.

[291] J. Paris and T. Arts. In *Proceedings of 8th ACM SIGPLAN workshop on ERLANG (Erlang'09), Edinburgh, Scotland, September 5, 2009*, pages 83–92. ACM, 2009.

[292] W. Perry. *Effective Methods for Software Testing*. John Wiley & Sons, Inc., 3rd edition, 2006.

[293] J. L. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice Hall PTR, 1981.

[294] M. Petre. UML in Practice. In *Proceedings of 35th International Conference on Software Engineering (ICSE'13), San Francisco, CA, USA, May 18-26 2013*, pages 722–731. IEEE Press, 2013.

[295] A. Petrenko. Why Automata Models Are Sexy for Testers? In *6th International Conference on Perspectives of Systems Informatics (PSI'06), Novosibirsk, Russia, June 27-30, 2006. Revised Papers*, volume 4378 of *Lecture Notes in Computer Science*, pages 26–26. Springer Berlin Heidelberg, 2007.

[296] L. Peyton, B. Stepien, and P. Seguin. Integration Testing of Composite Applications. In *Proceedings of 41st Annual Hawaii International Conference on System Sciences (HICSS'08), Waikoloa, Big Island, Hawaii, January 7-10, 2008*, pages 96–96. IEEE Computer Society, 2008.

[297] S. Pfleeger and J. Atlee. *Software Engineering: Theory and Practice*. Pearson, 2010.

[298] J. Piesing. The DVB multimedia home platform – "MHP". *IEE Colloquium on Interactive Television (Ref. No. 1999/200)*, pages 2/1–2/6, IET, 1999.

[299] J. Piesing. The DVB Multimedia Home Platform (MHP) and Related Specifications. *Proceedings of the IEEE*, 94(1):237–247, IEEE, Jan. 2006.

[300] T. M. Pigoski. *Practical Software Maintenance: Best Practices for Managing your Software Investment*. John Wiley & Sons, 1997.

[301] R. Pressman. *Software Engineering: A Practitioner's Approach*. McGraw-Hill, Inc., 7th edition, 2010.

[302] I. A. Qureshi and A. Nadeem. GUI Testing Techniques: A Survey. *International Journal of Future Computer and Communication*, 2(2):142–146, Apr. 2013.

[303] B. S. R. Probert and P. Xiong. Formal Testing of Web Content using TTCN-3. In *ETSI TTCN-3 User Conference, Sophia Antipolis, France, June 6-8, 2005*.

[304] D. M. Rafi, K. R. K. Moses, K. Petersen, and M. V. Mäntylä. Benefits and Limitations of Automated Software Testing: Systematic Literature Review and Practitioner Survey. In *Proceedings of 7th International Workshop on Automation of Software Test (AST'12), Zurich, Switzerland, June 2-3, 2012*, pages 36–42. IEEE Press, 2012.

[305] C. Rankin. The Software Testing Automation Framework. *IBM Syst. J.*, 41(1):126–139, IBM Corp., Jan. 2002.

[306] S. Rapps and E. J. Weyuker. Selecting Software Test Data Using Data Flow Information. *IEEE Transactions on Software Engineering*, 11(4):367–375, IEEE Press, Apr. 1985.

[307] M. W. Raza. Comparison of class test integration ordering strategies. In *Proceedings of IEEE Symposium on Emerging Technologies (ICET'05), Islamabad, Pakistan, September 17-18, 2005*, pages 440–444. IEEE, 2005.

[308] L. Richardson and S. Ruby. *Restful Web Services*. O'Reilly, 2007.

[309] L. Rising and N. S. Janoff. The Scrum Software Development Process for Small Teams. *IEEE Software*, 17(4):26–32, IEEE Computer Society Press, July 2000.

[310] S. Rivas. Developing a Set Top Box Middleware in Erlang. In *Erlang Factory, London, June 22-26, 2009*.

[311] S. Rivas, M. A. Francisco, and V. M. Gulías. Property Driven Development in Erlang, by Example. In *Proceedings of 5th Workshop on Automation of Software Test (AST'10), Cape Town, South Africa, May 1-8, 2010*, pages 75–78. ACM, 2010.

[312] E. M. Rodrigues, R. S. Saad, F. M. Oliveira, L. T. Costa, M. Bernardino, and A. F. Zorzo. Evaluating Capture and Replay and Model-based Performance Testing Tools: An Empirical Comparison. In *Proceedings of 8th ACM/IEEE International Symposium on Empirical Software Engineering and Measurement (ESEM'14), Torino, Italy, September 18-19, 2014*, pages 9:1–9:8. ACM, 2014.

[313] W. W. Royce. Managing the Development of Large Software Systems: Concepts and Techniques. In *Proceedings of 9th International Conference on Software Engineering (ICSE'87), Monterey, California, USA, 1987*, pages 328–338. IEEE Computer Society Press, 1987.

[314] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, and W. Lorensen. *Object-oriented Modeling and Design*. Prentice-Hall, Inc., 1991.

[315] J. Rumbaugh, I. Jacobson, and G. Booch. *The Unified Modeling Language Reference Manual*. Pearson Higher Education, 2nd edition, 2004.

[316] P. A. Salas and B. K. Aichernig. Automatic Test Case Generation for OCL: a Mutation Approach. Technical Report 321, The United Nations University - International Institute for Software Technology, 2005.

[317] P. S. Sandhu, P. P. Singh, and A. K. Verma. Evaluating Quality of Software Systems by Design Patterns Detection. In *Proceedings of International Conference on Advanced Computer Theory and Engineering (ICACTE'08), Phuket, Thailand, December 20-22, 2008*, pages 3–7. IEEE Computer Society, 2008.

[318] I. Schieferdecker and B. Stepien. Automated Testing of XML/SOAP based Web Services. In *ITG/GI-Fachtagung Kommunikation in Verteilten Systemen (KiVS'03), Leipzig, February 25–28, 2003*, Informatik aktuell, pages 43–54. Springer Berlin Heidelberg, 2003.

[319] I. Schieferdecker, B. Stepien, and A. Rennoch. PerfTTCN, a TTCN Language Extension for Performance Testing. In *10th International Workshop on Testing of Communicating Systems (IFIP TC6), Cheju Island, Korea, September 8–10, 1997*, pages 21–36. Springer US, 1997.

[320] W. Schütz. Fundamental Issues in Testing Distributed Real-time Systems. *Real-Time Systems*, 7(2):129–157, Kluwer Academic Publishers, Sept. 1994.

[321] K. Schwaber and M. Beedle. *Agile Software Development with Scrum*. Prentice Hall PTR, 2001.

[322] K. D. Schwartz. Automated Software Testing. Technical report, Information Week, 2003.

[323] M. Shafique and Y. Labiche. A Systematic Review of Model Based Testing Tool Support. Technical Report SCE-10-04, Carleton University, May 2010.

[324] M. Shaw and D. Garlan. Formulations and Formalisms in Software Architecture. In *Computer Science Today*, volume 1000 of *Lecture Notes in Computer Science*, pages 307–323. Springer Berlin Heidelberg, 1995.

[325] C. U. Smith. *Performance Engineering of Software Systems*. Addison-Wesley Longman Publishing Co., Inc., 1990.

[326] Software Engineering Institute. *Standard CMMI Appraisal Method for Process Improvement (SCAMPI), Version 1.3: Method Definition Document. Handbook CMU/SEI-2011-HB-001*. Carnegie Mellon University, Mar. 2011. http://www.sei.cmu.edu/reports/11hb001.pdf.

[327] C. Soldani. QuickCheck++. http://software.legiasoft.com/, Apr. 2015.

[328] I. Sommerville. *Software Engineering*. Addison-Wesley Publishing Company, 9th edition, 2007.

[329] I. Sommerville and G. Kotonya. *Requirements Engineering: Processes and Techniques*. John Wiley & Sons, Inc., 1998.

[330] I. Sommerville and P. Sawyer. *Requirements Engineering: A Good Practice Guide*. John Wiley & Sons, Inc., 1st edition, 1997.

[331] J. Stapleton. *DSDM, Dynamic Systems Development Method: The Method in Practice*. Addison-Wesley, 1997.

[332] B. Stepien and L. Peyton. A comparison between TTCN-3 and Python. In *TTCN-3 User Conference, Madrid, Spain, June 3-6, 2005*.

[333] B. Stepien and L. Peyton. Challenges of testing periodic messages in avionics systems using TTCN-3. In *Proceedings of 25th IFIP WG 6.1 International Conference (ICTSS'13), Istanbul, Turkey, November 13-15, 2013*, volume 8254 of *Lecture Notes in Computer Science*, pages 207–222. Springer Berlin Heidelberg, 2013.

[334] B. Stepien, L. Peyton, and P. Xiong. Framework testing of web applications using TTCN-3. *International Journal on Software Tools for Technology Transfer*, 10(4):371–381, Springer-Verlag, Aug. 2008.

[335] N. R. Storey. *Safety Critical Computer Systems*. Addison-Wesley Longman Publishing Co., Inc., 1996.

[336] B. Subramaniam. Effective Software Defect Tracking. Reducing Project Costs and Enhancing Quality. *CrossTalk: The Journal of Defense Software Engineering*, 12(4):3–9, Apr. 1999.

[337] B. M. Subraya. *Integrated Approach to Web Performance Testing: A Practitioner's Guide*. IRM Press, 2006.

[338] Sun Microsystems, Inc. *Java TV API 1.1 (JSR-927)*, 2006.

[339] M. Sutton, A. Greene, and P. Amini. *Fuzzing: Brute Force Vulnerability Discovery*. Addison-Wesley Professional, 2007.

[340] C. Szyperski. *Component Software: Beyond Object-Oriented Programming*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2002.

[341] P. Tahchiev, F. Leme, V. Massol, and G. Gregory. *JUnit in Action*. Manning Publications Co., 2nd edition, 2010.

[342] A. Takanen, J. DeMott, and C. Miller. *Fuzzing for Software Security Testing and Quality Assurance*. Artech House, Inc., 2008.

[343] G. Tassey. The economic impacts of inadequate infrastructure for software testing. Technical Report 02-3, National Institute of Standards and Technology, May 2002.

[344] R. Thayer and M. Dorfman. *Software Requirements Engineering*. Wiley-IEEE Press, 1997.

[345] P. Thévenod-Fosse, H. Waeselynck, and Y. Crouzet. An Experimental Study on Software Structural Testing: Deterministic versus Random Input Generation. In *Proceedings of 21st International Symposium on Fault-Tolerant Computing (FTCS'91), Montreal, Quebec, Canada, June 25-27, 1991*, pages 410–417. IEEE Computer Society, 1991.

[346] J. Timm and G. Gannod. Specifying semantic web service compositions using UML and OCL. In *Proceedings of IEEE International Conference on Web Services (ICWS'07), Salt Lake City, UT, July 9-13, 2007*, pages 521–528. IEEE, 2007.

[347] R. Torkar and S. Mankefors. A Survey on Testing and Reuse. In *Proceedings of IEEE International Conference on Software: Science, Technology & Engineering (SWSTE'03), Herzlia, Israel, November 4-5, 2003*, pages 164–173. IEEE Computer Society, 2003.

[348] M. Utting and B. Legeard. *Practical Model-Based Testing: A Tools Approach*. Morgan Kaufmann, 2007.

[349] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *The Journal of Software Testing, Verification and Reliability*, 22(5):297–312, John Wiley & Sons, Ltd., Aug. 2012.

[350] T. Van Tongeren. A Tester's Tale. *Software Magazine*, 20(2):46, Apr. 2000.

[351] D. Vega, G. Din, and I. Schieferdecker. Application of TTCN-3 test language to testing information systems in eHealth domain. In *Proceedings of International Conference on Multimedia Computing and Information Technology (MCIT'10), Sharjah, UAE, March 2-4, 2010*, pages 21–24. IEEE, 2010.

[352] J. M. Verner, J. Sampson, and N. Cerpa. What factors lead to software project failure? In *Proceedings of 2nd International Conference on Research Challenges in Information Science (RCIS'08), Marrakech, Morocco, June 3-6, 2008*, pages 71–80. IEEE, 2008.

[353] W3C. WSDL – Web Services Description Language. http://www.w3.org/TR/wsdl, Mar. 2001.

[354] W3C. Web Services Architecture. http://www.w3.org/TR/ws-arch, Feb. 2004.

[355] W3C. WSDL-S – Web Services Semantics. http://www.w3.org/Submission/WSDL-S, Nov. 2005.

[356] W3C. SOAP Specifications. http://www.w3.org/TR/soap, Apr. 2007.

[357] W3C. XML – eXtensible Markup Language. http://www.w3.org/TR/xml, Nov. 2008.

[358] W3C. WADL – Web Application Description Language. http://www.w3.org/Submission/wadl, Aug. 2009.

[359] W3C. XSD – W3C XML Schema Definition Language – Part 1: Structures. http://www.w3.org/TR/xmlschema11-1, Apr. 2012.

[360] W3C. XSD – W3C XML Schema Definition Language – Part 2: Datatypes. http://www.w3.org/TR/xmlschema11-2, Apr. 2012.

[361] F. Wagner, R. Schmuki, T. Wagner, and P. Wolstenholme. *Modeling Software with Finite State Machines: A Practical Approach*. Auerbach Publications, 2006.

[362] D. Wang, J. Kuang, and W. Tan. Conformance testing for the car lights system based on AUTOSAR standard. In *Proceedings of IEEE 3rd International Conference on Communication Software and Networks (ICCSN'11), Xi'an, China, May 27-29, 2011*, pages 345–348. IEEE, 2011.

[363] J. Warmer and A. Kleppe. *The Object Constraint Language: Getting Your Models Ready for MDA*. Addison-Wesley Longman Publishing Co., Inc., 2nd edition, 2003.

[364] G. M. Weinberg. *Quality Software Management (Vol. 1): Systems Thinking*. Dorset House Publishing Co., Inc., 1992.

[365] J. A. Whittaker. *Exploratory Software Testing: Tips, Tricks, Tours, and Techniques to Guide Test Design*. Addison-Wesley Professional, 2009.

[366] K. Wiegers and J. Beatty. *Software Requirements*. Microsoft Press, 3rd edition, 2013.

[367] C. Willcock, T. Deiß, S. Tobies, S. Keil, F. Engler, and S. Schulz. *An Introduction to TTCN-3*. Wiley, 2nd edition, 2011.

[368] R. Wirfs-Brock, B. Wilkerson, and L. Wiener. *Designing Object-oriented Software*. Prentice-Hall, Inc., 1990.

[369] N. Wirth. Program Development by Stepwise Refinement. *Communications of the ACM*, 14(4):221–227, ACM, Apr. 1971.

[370] Y. Wu, M. Chen, and J. Offutt. UML-based integration testing for component-based software. In *Proceedings of 2nd International Conference on COTS-Based Software Systems (ICCBSS'03), Ottawa, Canada, February 10–12, 2003*, volume 2580 of *Lecture Notes in Computer Science*, pages 251–260. Springer Berlin Heidelberg, 2003.

[371] P. Xiong, R. L. Probert, and B. Stepien. An efficient formal testing approach for web service with TTCN-3. In *Proceedings of 13th International Conference on Software, Telecommunications and Computer Networks (SoftCOM'05), Split, Marina Frapa, Croatia, September 15-17, 2005*.

[372] S. S. Yau and J. J.-P. Tsai. A Survey of Software Design Techniques. *IEEE Transactions on Software Engineering*, 12(6):713–721, IEEE Press, June 1986.

[373] E. Yourdon and L. L. Constantine. *Structured Design: Fundamentals of a Discipline of Computer Program and Systems Design*. Prentice-Hall, Inc., 1st edition, 1979.

[374] A. Zeller and R. Hildebrandt. Simplifying and Isolating Failure-Inducing Input. *IEEE Transactions on Software Engineering*, 28(2):183–200, IEEE Press, Feb. 2002.

[375] Y. Z. Zhang, W. Fu, and J. Y. Qian. Automatic testing of web services in Haskell platform. *Journal of Computational Information Systems*, 6(9):2859–2867, 2010.

[376] Y. Zheng, J. Zhou, and P. Krause. An automatic test case generation framework for web services. *Journal of Software*, 2(3):64–77, Sept. 2007.