novática
Revista de la Asociación de Técnicos de Informática

Novática, revista fundada en 1975 y decana de la prensa informática española, es el órgano oficial de expresión y formación continua de **ATI** (Asociación de Técnicos de Informática), organización que edita también la revista **REICIS** (Revista Española de Innovación, Calidad e Ingeniería del Software).

<http://www.ati.es/novatica/>
<http://www.ati.es/reicis/>

ATI es miembro fundador de **CEPIS** (*Council of European Professional Informatics Societies*) y es representante de España en **IFIP** (*International Federation for Information Processing*); tiene un acuerdo de colaboración con **ACM** (*Association for Computing Machinery*), así como acuerdos de vinculación o colaboración con **AdaSpain**, **AI2**, **ASTIC**, **RITSI** e **Hispalinux**, junto a la que participa en **ProInnova**.

Coordinación Editorial, Redacción Central y Redacción ATI Madrid
Plaza de España 6, 2ª planta, 28008 Madrid
Tlfn.914029391; fax.913093685 <novatica@ati.es>
Composición, Edición y Redacción ATI Valencia
Av. del Reino de Valencia 23, 46005 Valencia
Tlfn. 963740173 <novatica_prod@ati.es>
Administración y Redacción ATI Cataluña
Calle Ávila 50, 3a planta, local 9, 08005 Barcelona
Tlfn.934125235; fax 934127713 <secregen@ati.es>
Redacción ATI Andalucía <secreand@ati.es>
Redacción ATI Galicia <secregal@ati.es>
Suscripción y Ventas <novatica.subscripciones@atinet.es>
Publicidad Plaza de España 6, 2ª planta, 28008 Madrid
Tlnf.914029391; fax.913093685 <novatica@ati.es>
Imprenta: Derra S.A., Juan de Austria 66, 08005 Barcelona.
Depósito legal: B 15.154-1975 – ISSN: 0211-2124; CODEN NOVAEC
Portada: "Sueños probatorios" - Concha Arias Pérez / © ATI
Diseño: Fernando Agresta / © ATI 2003

sumario

Nº 224, julio-agosto 2013, año XXXIX

Tema del próximo número: "Empresa 2.0"

Daniel Rodríguez García[1], José Javier Dolado Cosín[2]
[1]Universidad de Alcalá, [2]UAH; Universidad del País Vasco, UPV/EHU; [1, 2]Coordinadores de la sección técnica "Ingeniería del Software" de Novática

<danrodgar@gmail.com>, <javier.dolado@ehu.es>

Presentación
Mejorando el proceso de pruebas de software: Estado del arte

Editores invitados

Daniel Rodríguez García es profesor titular del Departamento de Ciencias de la Computación de la Universidad de Alcalá, licenciado en Informática por la Universidad del País Vasco/Euskal Herriko Unibertsitatea, y doctorado por la Universidad de Reading. Sus intereses se centran en la ingeniería del software, y la aplicación de técnicas de minería de datos y optimización a la ingeniería del software.

José Javier Dolado Cosín es Catedrático de Universidad en el Departamento de Lenguajes y Sistemas Informáticos de la Universidad del País Vasco/Euskal Herriko Unibertsitatea. Sus intereses técnicos giran alrededor de la ingeniería del software, los sistemas complejos y la gestión de proyectos.

La monografía que presentamos a continuación está dedicada a las "pruebas de software". Las pruebas de software constituyen una de las etapas de desarrollo de software que menos interés recibe cuando se define un proyecto de desarrollo de software a pesar de que todo gestor de proyectos es consciente de la importancia de las mismas.

Siendo las actividades de desarrollo más atractivas desde el punto de vista de la creación de aplicaciones, la gestión de las pruebas conlleva un conjunto de actividades no menos interesantes e igual de complejas. Por ejemplo, Panagiotis Louridas [1] menciona algunas de estas tareas, tales como:

- Organizar el conjunto de casos de prueba.
- Asignar los casos de prueba a los responsables de las pruebas.
- Dirigir el proceso de pruebas.
- Recoger los resultados.
- Medir el progreso de las pruebas.

Para llevar a cabo estas tareas disponemos de diversas herramientas y procesos. El número de herramientas de ayuda a la realización de las pruebas es cada vez mayor, disponiendo los desarrolladores de múltiples útiles para probar el código a medida que se va creando. Del mismo modo, también se incrementan las propuestas de procesos o modos de gestionar y realizar las pruebas. Términos como "*test-driven development*" o "*agile testing*" [2] ya son habituales dentro del desarrollo de software y los equipos de desarrollo se adecuan a los nuevos modos de trabajo en los que se integran las pruebas.

También aparecen nuevos enfoques en el desarrollo de software que modificarán nuevamente los modos de realizar las pruebas. Así, recientemente se han descrito los efectos que pueden tener en las pruebas la aplicación de "las reglas de negocio" [3].

Un desarrollo basado en reglas de negocio conlleva, de acuerdo con sus proponentes, una reducción en los problemas de comprensión entre los grupos de "*testers*" y otros grupos de trabajo. También se indica que, debido a la visibilidad de las reglas de negocio, aumenta la comprensión del comportamiento de los programas y permite escribir mejores escenarios de pruebas. Éste es un ejemplo más de cómo el área de las pruebas de software se ve afectada por los modos de desarrollo.

Según los datos publicados en [1] el esfuerzo dedicado a las pruebas (específicamente "*system test*") varía en función del tamaño del proyecto medido en KLOC (miles de líneas de código), dedicando desde un 16% de esfuerzo en proyectos con un tamaño de 1 KLOC hasta llegar a un 29% para proyectos de 500 KLOC.

También el dominio de aplicación es importante, puesto que el ratio nº de "*developers*"/ nº de "*testers*" puede variar desde 20:1 hasta 1:10. El valor más bajo (20:1) corresponde a dominios de sistemas de gestión comerciales y el valor más alto corresponde a los sistemas de control de vuelo de la NASA. Así, tamaño y dominio de la aplicación son factores a considerar junto con los modelos de procesos de desarrollo y de pruebas.

Pero el término que hoy día se escucha por doquier, "*cloud computing*" y que, quizá, cambie los modos de entender la Informática, también tiene sus consecuencias en el modo de entender las pruebas del software. Así lo expresa J. Whittaker, director de ingeniería de Google, que se pregunta sobre los modos en los que las pruebas unitarias, las pruebas de integración y las pruebas de sistema se adaptan al nuevo paradigma de computación en la nube [4][5].

Para alguna de las actividades, estos autores sugieren realizar unos planes de pruebas con una duración de 10 minutos, pero ir modificando la aplicación objetivo de manera continuada. Según su experiencia [5], la concentración de los equipos mejoró y, aunque no se consiguió llegar al límite de los 10 minutos, los equipos consiguieron definir un plan de pruebas en 30 minutos, incluyendo la documentación esencial. Como ellos mismos indican, su experiencia no tiene por qué ser válida para entornos con software de alta seguridad pero es un ejemplo de cómo se puede desarrollar el núcleo de una planificación de las pruebas en muy poco tiempo.

Por otra parte, la realización de las pruebas de software no tiene por qué ser únicamente la actividad del grupo de *testers* tal como las evidencias lo indican en algunas organizaciones.

Así, Mika V. Mäntylä et al. [6] encontraron evidencias en varias organizaciones de que las pruebas no eran una acción realizada únicamente por los especialistas en las mismas. Los miembros de los equipos de trabajo que tenían contacto con los usuarios y clientes mostraron una alta tasa de validación al igual que la tenían los desarrolladores que encontraban defectos. La conclusión de este estudio es que es importante reconocer la diversidad de individuos que realmente participan en las pruebas, y también es importante comprender la relevancia de la validación desde el punto de vista de usuario final.

Desde un punto de vista más técnico, es de destacar el auge de las técnicas de búsqueda aplicadas a la Ingeniería del Software en general y *testing* en particular aplicando técnicas metaherísticas.

Mark Harman et al. [7] han publicado recientemente el estado de la cuestión en este área y mantienen un repositorio con publicaciones relacionadas[1].

La literatura sobre las pruebas de software es amplísima y es imposible abarcar todos los aspectos en un número limitado de páginas. En la monografía que hoy presentamos, encontramos varios artículos escritos por autores punteros en diversas áreas de pruebas de software.

El trabajo realizado por ***Manuel Núñez***, ***Mercedes G. Merayo*** y ***Robert M. Hierons***,

titulado "*Procesos de pruebas basados en modelos: un compromiso adecuado entre teoría y práctica*", aborda la cuestión de la utilización de los modelos formales dentro del proceso de pruebas.

Los autores ***Javier Tuya***, ***Claudio de la Riva***, ***María Jose Suárez-Cabal*** y ***Raquel Blanco*** presentan en su artículo "*Cobertura de consultas SQL y sus aplicaciones*" las cuestiones más relevantes sobre la cobertura de consultas SQL y su aplicación en las pruebas de bases de datos.

En el artículo "*Algoritmos bio-inspirados para la automatización de pruebas software en la industria*", los autores ***Javier Ferrer***, ***Francisco Chicano*** y ***Enrique Alba*** abordan la aplicación de algoritmos de inspiración biológica a las pruebas del software.

El cuarto trabajo de la sección titulado "*Priorización de casos de prueba: Avances y retos*" y realizado por ***Ana Belén Sánchez Jerez***, ***Sergio Segura Rueda*** y ***Antonio Ruiz-Cortés***, presenta una clasificación para la priorización de los casos de prueba.

El último artículo de esta primera sección de la monografía, "*Utilización de MDE para la Prueba de Sistemas de Información Web*" y escrito por ***Federico Toledo Rodríguez***, ***Macario Polo Usaola*** y ***Beatriz Pérez Lamancha***, describe las cuestiones esenciales en la aplicación de la ingeniería dirigida por los modelos a las pruebas de software.

Por último, tenemos la oportunidad de conocer de primera mano, gracias a la reseña escrita por ***Javier Tuya***, los primeros detalles sobre la *norma ISO/IEC/IEEE 29119 - Software Testing.*

El conjunto de artículos que hemos mencionado hasta ahora, cubre una de la parte de las áreas de pruebas del software que en los próximos años tendrán un importante desarrollo.

Por otro lado, este número especial también cuenta con aportaciones desde un punto de vista industrial. ***Tanja E. Vos***, ***Beatriz Marín*** y ***María José Escalona Cuaresma*** presentan un marco metodológico para evaluar ambas técnicas y herramientas de pruebas con la idea de ayudar a las empresas a decidir qué pruebas deben usarse en cada contexto y cómo comparar las distintas herramientas disponibles.

Finalmente, ***Celestina Bianco*** nos presenta un caso de estudio en una industria farmacéutica de la aplicación de métricas para el apoyo de la toma de decisiones en las pruebas de un dispositivo de análisis de sangre para cumplir así con los objetivos de calidad y estándares necesarios.

Referencias

[1] P. Louridas. Test Management. *IEEE Software, 87*, Sept/Oct 2011, pp. 86-91.

[2] L. Crispin, J. Gregory. *Agile Testing: A Practical Guide for Testers and Agile Teams*. Pearson, 2009. ISBN-10: 0321534468.

[3] T. O. Meservy, C. Zhang, E.T. Lee, J. Dhaliwal. The Business Rules Approach and its Effect on Software Testing. *IEEE Software, July/August, 2012*, pp. 60-66.

[4] F. Shull, A Brave. New World of Testing? An Interview with Google's James Whittaker. *IEEE Software, March/April, 2012*, pp. 4-7.

[5] J.A. Whittaker. The 10-Minute Test Plan. *IEEE Software, November/December 2012*, pp. 70-77.

[6] M.V. Mäntylä, J. Itkonen, J. Iivonen. Who tested my software? Testing as an organizationally cross-cutting activity. *Software Quality Journal, 20*:pp. 145-172 (2012). Descargable en PDF desde: <https://wiki.aalto.fi/pages/viewpage.action?pageId=58940614>. Último acceso: 30 de octubre de 2013.

[7] M. Harman, S.A. Mansouri, Y. Zhang. Search-based software engineering: Trends, techniques and applications. *ACM Computing Surveys, 45 (1)*, Artículo 11 (diciembre 2012).

Nota

[1] <http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/>.

Procesos de pruebas basados en modelos: Un compromiso adecuado entre teoría y práctica

Manuel Núñez[1], Mercedes G. Merayo[1], Robert M. Hierons[2]
[1]Departamento de Sistemas Informáticos y Computación, Universidad Complutense de Madrid; [2]Department of Information Systems and Computing, Brunel University Uxbridge, Middlesex (Reino Unido)

<mn@sip.ucm.es>,<mgmerayo@fdi.ucm.es>, <rob.hierons@brunel.ac.uk>

Resumen: *En este artículo presentaremos nuestras contribuciones más recientes en el campo de los procesos de pruebas basados en modelos. Revisaremos las principales características de nuestro trabajo pero omitiremos todos los detalles técnicos. Concluiremos el artículo con una breve discusión sobre la utilidad de formalizar, en la mayor medida posible, los procesos de pruebas y enumerando distintas formas en las que nuestros modelos y resultados se pueden aplicar en entornos industriales.*

Palabras clave: *Procesos de pruebas basados en modelos, sistemas distribuidos, sistemas temporizados.*

Autores

Manuel Núñez es Catedrático de Universidad en la Facultad de Informática de la Universidad Complutense de Madrid, España. Su investigación se centra en el estudio de métodos formales, con un énfasis especial en aproximaciones formales a los procesos de pruebas. Ha participado en más de 100 Comités de Programa y ha publicado más de 130 trabajos en revistas y conferencias internacionales. <http://antares.sip.ucm.es/manolo/>.

Mercedes G. Merayo es Profesora Titular de Universidad en la Facultad de Informática de la Universidad Complutense de Madrid. Su investigación se centra en procesos de pruebas basados en modelos, con un énfasis especial en el análisis de sistemas temporizados. Ha participado en más de 40 Comités de Programa y ha publicado más de 50 trabajos en revistas y conferencias internacionales. <http://antares.sip.ucm.es/mercedes/>.

Robert M. Hierons es *Professor of Computing* en la *School of Information Systems, Computing and Mathematics* de la Universidad de Brunel (Reino Unido). Su investigación se centra en procesos de pruebas basados en modelos, con un énfasis especial en el estudio de sistemas distribuidos, sistemas en tiempo real y sistemas asíncronos. Ha participado en más de 100 Comités de Programa y ha publicado más de 190 trabajos en revistas y conferencias internacionales. <http://www.brunel.ac.uk/~csstrmh/>.

1. Introducción

La creciente complejidad de los sistemas actuales dificulta notablemente la tarea de asegurar altos niveles de confianza en la corrección de las aplicaciones desarrolladas. Este hecho queda patente, con mayor intensidad, en los sistemas software que se usan en áreas donde la seguridad de las personas es un factor crítico (por ej. sistemas de control de tráfico aéreo, centrales nucleares, sistemas de control médico, etc.).

Para paliar esta situación, algunas grandes compañías están tratando de incrementar el grado de formalización a la hora de desarrollar sus sistemas. Por ejemplo, Schneider-Electric y Airbus utilizan el entorno de desarrollo SCADE, basado en el lenguaje formal LUSTRE, para crear software de control de centrales nucleares y el software de control de los modelos Airbus A340/600 y A380, respectivamente. Merece mención aparte Microsoft que durante los últimos años ha puesto un énfasis especial en la aplicación de métodos formales en la producción de su software.

Entre las distintas técnicas utilizadas para incrementar el grado de confianza en la corrección de los sistemas desarrollados podemos afirmar que los *procesos de pruebas* representan la técnica más extendida en entornos industriales.

Brevemente, se puede decir que un proceso de pruebas consiste en comprobar, mediante interacciones sucesivas, que un sistema cumple un conjunto de propiedades y comportamientos deseados. Este tipo de procesos se ha convertido en una parte fundamental del ciclo de vida. De hecho, se ha constituido en un área de trabajo muy próspera que cuenta con la participación activa de una importante comunidad de investigadores y expertos.

Durante la última década se ha implantado una fuerte consciencia sobre la necesidad de aplicar técnicas formales en los procesos de pruebas para conseguir una mayor automatización y, en último término, obtener resultados más fiables. A pesar de todo ello, los procesos de pruebas se realizan generalmente de un modo *artesanal*, decreciendo así los beneficios que podrían obtenerse de la aplicación formal y sistemática de estas técnicas.

Aunque existen numerosas propuestas rigurosas, el alto nivel de abstracción de estos modelos usualmente limita su implantación en entornos industriales. Para poder contrarrestar esta situación, la comunidad científica está realizando un esfuerzo para desarrollar herramientas que implementen las metodologías formales de los procesos de pruebas. Además, algunas compañías punteras dedicadas a la creación de software han introducido en sus procesos de producción propuestas formales para realizar los procesos de pruebas en los sistemas que desarrollan. Entre ellas merece la pena mencionar de nuevo el caso de Microsoft, que mantiene un amplio grupo trabajando en métodos formales para los procesos de pruebas.

En esta línea es especialmente destacable el trabajo [1], en el que investigadores de Microsoft mostraron que el uso de métodos formales aumenta la fiabilidad de los sistemas desarrollados y permite ahorrar en el presupuesto dedicado a los procesos de pruebas, incluso en el caso de utilizar empleados que no tenían conocimientos previos de métodos formales.

Una manera habitual de aumentar la formalización de los procesos de pruebas consiste en utilizar un modelo (usualmente parcial) del sistema a desarrollar y realizar las pruebas partiendo de la información que nos proporciona dicho modelo (el trabajo recopilatorio [2] permite obtener una amplia visión de este campo).

Intuitivamente, los procesos de pruebas basados en modelos consisten en aplicar pruebas al sistema desarrollado y comprobar si el resultado observado es coherente con lo que el modelo establece que debería ocurrir en la situación planteada. Habitualmente estos procesos de pruebas asumen que el sistema que se está analizando es una *caja negra*, es decir, no es posible acceder a su estructura interna (en el caso del software, no se dispone del código fuente) sino que el proceso de

> “Una manera habitual de aumentar la formalización de los procesos de pruebas consiste en utilizar un modelo (usualmente parcial) del sistema a desarrollar y realizar las pruebas partiendo de la información que nos proporciona dicho modelo”

pruebas debe limitarse a interaccionar con el sistema.

Una representación esquemática de estos procesos se presenta en la **figura 1**.

Aunque los procesos de pruebas basados en modelos representan un campo bien establecido, con numerosas propuestas alternativas y con herramientas que sirven para apoyar el uso de las metodologías desarrolladas, la mayoría del trabajo se ha centrado en determinar que el sistema hace de forma correcta lo que se supone que debe hacer.

Sin embargo, en muchos sistemas es especialmente importante asegurarnos no solo de que el sistema hace lo que debe hacer, sino de que lo hace de una determinada forma. En este sentido, merece especial atención el estudio de comportamientos en los que el tiempo juega un papel fundamental.

De esta forma, debemos no solo determinar que el sistema realiza una cierta acción sino que la realiza, por ejemplo, antes de que transcurra una cierta cantidad de tiempo.

En este artículo revisaremos nuestras contribuciones en el campo de los procesos de pruebas basados en modelos para sistemas con información temporal.

Otra de las líneas en la que estamos trabajando, dentro del amplio campo de los procesos de pruebas basados en modelos, consiste en aplicar los modelos clásicos a sistemas con componentes distribuidos. Estos sistemas presentan unas peculiaridades que, desde el punto de vista teórico, se pueden resumir en el hecho de que si no contamos con un centralizador con el que todas las componentes se comuniquen constantemente, entonces no es posible conocer el orden exacto en el que las acciones se produjeron.

El resto de este artículo está estructurado de la siguiente forma. En la **sección 2** presentamos nuestros trabajos en procesos de pruebas para sistemas donde su comportamiento temporal debe ser tenido en cuenta. En la **sección 3** revisamos nuestras contribuciones en el marco de los procesos de pruebas de sistemas que tienen sus componentes distribuidos entre distintas ubicaciones. En la **sección 4** presentamos nuestro trabajo en procesos de pruebas de forma pasiva.

Intuitivamente, un proceso de pruebas es *pasivo* si en lugar de interaccionar con el sistema, se observan los comportamientos de dicho sistema para determinar si son acordes a las propiedades representadas en sus requisitos. Finalmente, en la **sección 5** presentamos nuestras conclusiones y algunas líneas de trabajo en las que se pueden aplicar nuestras metodologías.

2. Procesos de pruebas en sistemas temporizados

2.1. Modelos teóricos

En la actualidad existen numerosos marcos para realizar procesos de pruebas basados en modelos sobre sistemas que presenten requisitos temporales.

Usualmente, estas metodologías utilizan una noción de tiempo *determinista* en el sentido de que los requisitos expresan condiciones de la forma "*después/antes de t unidades de tiempo*".

Sin embargo, en muchas situaciones no es adecuado utilizar una noción tan simple para representar propiedades temporales. En esta línea, la utilización de modelos estocásticos permite especificar propiedades tales como "*con probabilidad p esta acción debería realizarse antes de t unidades de tiempo*". De este modo, el especificador no debe indicar un tiempo exacto para una acción determinada sino una estimación probabilística.

Cuando no se dispone de esta información probabilística, o bien se considera que su inclusión complicaría innecesariamente el

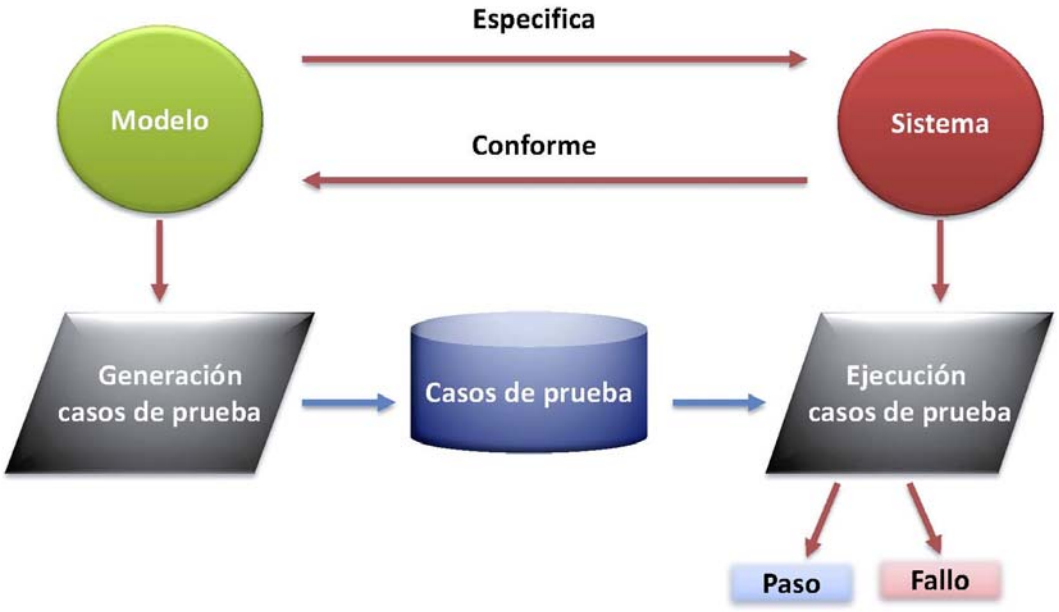

Figura 1. Esquema de proceso de pruebas *activo*.

> “En este punto es donde nuestra metodología para realizar procesos de pruebas basados en modelos con información temporal entra en juego”

modelo, la forma más apropiada para especificar restricciones temporales es el uso de *intervalos de tiempo* que indican el conjunto de posibles valores que se puede asociar con la ejecución de una cierta acción.

Estas disquisiciones han dado lugar al desarrollo de técnicas que se ajustan a las diferentes nociones temporales que se pueden considerar. Desgraciadamente, en la mayoría de los casos dichas técnicas solo son aplicables en un dominio temporal específico, no siendo sencilla su adaptación a sistemas donde se utilizan otros dominios temporales.

En [3] se presenta un marco unificado para especificar y analizar sistemas donde los requisitos temporales se pueden expresar en tres dominios diferentes: *tiempos fijos, variables aleatorias e intervalos de tiempo*.

En primer lugar, es necesario establecer con precisión las condiciones bajo las que se considera que una implementación es correcta con respecto a una especificación. En el caso de modelos deterministas, la noción estándar de corrección es la *equivalencia*: la implementación debe mimetizar el comportamiento de la especificación. Sin embargo, en el caso de que consideremos sistemas no deterministas es más apropiado utilizar una noción de corrección más débil, usualmente denominada *conformidad,* que esencialmente consiste en asegurar que el sistema implementado no *inventa* comportamientos que no estuvieran previstos en la especificación.

En este trabajo se definen varias *relaciones de conformidad*. Todas estas relaciones exigen que el sistema implementado, sin tener en cuenta los aspectos temporales, sea *conforme* con la especificación.

Además, se requiere el cumplimiento de diferentes condiciones temporales, permitiendo que se pueda elegir la más apropiada en cada caso. Por ejemplo, una de las relaciones establece que los tiempos consumidos por la implementación para realizar las acciones sean siempre menores que los indicados en la especificación, mientras en otro caso se exige que sean exactamente los mismos.

Como ya hemos apuntado anteriormente, en el caso de usar un marco temporal *estocástico*, la información se define en términos probabilísticos: la duración de las acciones se expresa en la especificación mediante *variables aleatorias*. Ello permite tener requisitos de la forma "*el tiempo de ejecución de la acción* a *está determinado por la distribución de la variable aleatoria* $\hat{i}_a$". En lo referente a la relación de conformidad para este tipo de sistemas, el hecho de asumir un marco de caja negra impide determinar si las variables aleatorias (desconocidas) asociadas con cada una las acciones del sistema que ha sido implementado están distribuidas de forma idéntica a las correspondientes en la especificación (conocidas). Por ello, es necesario plantear una relación de conformidad más *apropiada*, basada en un conjunto finito de observaciones. La idea consiste en comprobar que los tiempos de ejecución observados son compatibles con las variables aleatorias que aparecen reflejadas en la especificación, estableciéndose dicha compatibilidad por medio de un *contraste de hipótesis*.

2.2. Propuesta metodológica

Hasta ahora hemos discutido la noción de corrección de un sistema con respecto a una especificación que presenta requisitos temporales. Sin embargo, es necesario plasmar en un marco realista los modelos teóricos subyacentes a las relaciones de conformidad. En este punto es donde nuestra metodología para realizar procesos de pruebas basados en modelos con información temporal entra en juego. Idealmente, debemos ser capaces de generar un conjunto de casos de prueba tales que un sistema es conforme con respecto a una especificación si y solo si pasa correctamente todos los casos de prueba.

En [3] presentamos un algoritmo de derivación de casos de prueba que tiene en cuenta los comportamientos que deben analizarse obligatoriamente. Nótese que si no se establece ningún criterio de cobertura, en general tendremos un conjunto infinito de casos de prueba, de forma que suele ser interesante complementar el marco general con técnicas heurísticas que permitan recoger conjuntos representativos de casos de prueba [4].

En un trabajo posterior [5] hemos estudiado en profundidad distintos marcos alternativos para especificar y analizar sistemas con requisitos temporales impuestos mediante intervalos. La principal peculiaridad de este extenso artículo, y que lo distingue del núcleo principal de trabajos en procesos de pruebas para sistemas temporizados, es que permite que un sistema presente *pequeños* errores en su comportamiento temporal. Por ejemplo, si nuestra especificación indica que una acción debe tardar en completarse más de un milisegundo pero menos de tres, y observamos un sistema durante mil interacciones de forma que todas menos dos de ellas caen en el intervalo especificado, entonces podríamos considerar que el sistema es correcto dado que la discrepancia es despreciable y podría deberse, en particular, a la imprecisión de los elementos utilizados para la medida del tiempo.

Por supuesto, serán los requisitos del sistema los que marquen si un error pequeño es admisible o si por el contrario, como en el caso de acciones críticas que tienen que tener un comportamiento exacto en todas las ocasiones, no se permite ninguna desviación del comportamiento esperado.

Los modelos mencionados anteriormente incluyen sólo una noción de paso del tiempo: el tiempo asociado con la ejecución de acciones por parte del sistema. Sin embargo, existen otras situaciones en las que es conveniente tener en cuenta el paso del tiempo cuando se evalúa el comportamiento del sistema. Esta consideración motivó la propuesta de un nuevo marco para llevar a cabo procesos de pruebas [6] en sistemas que pueden evolucionar por medio de *timeouts*.

Esencialmente, un *timeout* es un periodo de tiempo específico durante el cual el sistema permanece a la espera de recibir un estímulo del entorno. Si este tiempo se rebasa y no se ha producido ninguna interacción con el sistema, éste cambia de estado, por lo que sus reacciones a los estímulos recibidos pueden ser diferentes a las que se hubieran producido con anterioridad a dicho límite de tiempo.

Esta consideración complica el marco teórico debido al impacto sobre el comportamiento del sistema, en lo que respecta a las acciones que realiza, de los posibles *timeouts*: una misma secuencia de entradas puede generar diferentes secuencias de salidas dependiendo de los *timeouts* que se hayan producido. Por lo tanto, los aspectos temporales del sistema afectan parcialmente a su comportamiento funcional.

3. Procesos de pruebas en sistemas altamente distribuidos

3.1. Problemática planteada

Los sistemas distribuidos están constituidos por diferentes componentes que se comunican con su entorno a través de diferentes puertos. La principal complicación que pre-

> “En nuestra propuesta hemos considerado que era asumible cuantificar las diferencias potenciales de tiempos entre los diferentes relojes locales y se han investigado diferentes escenarios que han dado lugar a diferentes relaciones de implementación”

sentan estos sistemas durante el proceso de pruebas consiste en integrar apropiadamente la información obtenida por los agentes que realizan las pruebas en cada uno de los puertos. Ello requiere que la interacción de los mismos con el sistema deba realizarse de forma coordinada.

Dado que los puertos pueden llegar a estar en diferentes ubicaciones geográficas, el único modo de sincronizar las interacciones de acuerdo a un plan centralizado requiere una infraestructura de comunicación entre los agentes.

Esta situación puede acarrear un alto coste e incluso interferir con el sistema que se está analizando, pudiendo afectar a los comportamientos observados. Por lo tanto, en lugar de considerar un proceso de pruebas centralizado, es necesario aplicar un plan de interacción para cada puerto en el que, en la medida de lo posible, solo se consideren las acciones que le corresponden y cuyos resultados proporcionen información relevante respecto a la corrección del sistema. Una representación esquemática de esta arquitectura se puede ver en la **figura 2**.

Debido a la encapsulación de estos procesos en cada puerto surgen problemas de *controlabilidad* y *observabilidad*.

El primero de estos problemas corresponde a la dificultad para decidir cuándo se deben aplicar las entradas en cada puerto, debido al desconocimiento de lo que ocurre en los demás puertos. Por ejemplo, supongamos que un caso de prueba comienza con la aplicación de una entrada x en el puerto P, seguida de una salida y en el mismo puerto; a continuación, el agente encargado del puerto Q debe aplicar la entrada z.

El problema que aparece es que desde el puerto Q no se pueden observar las interacciones en el puerto P y, por tanto, no es posible determinar cuándo se debe aplicar la entrada z.

El problema de observabilidad hace referencia a la dificultad para determinar si las observaciones independientes en cada puerto corresponden a un comportamiento correcto del sistema. Esta situación puede llevar a un fallo enmascarado. Supongamos que, como en el ejemplo anterior, un caso de prueba comienza con la aplicación de la entrada x y la emisión de una salida y en el puerto P ; a continuación se debe aplicar nuevamente x y observar la salida y en el mismo puerto P, así como la salida z en el puerto Q. La secuencia inducida por el caso de prueba en el puerto P es $xyxy$ mientras que en el puerto Q deberíamos observar z.

Nótese que las secuencias observadas en ambos puertos serían las mismas si las salidas y y z se produjesen como respuesta a la primera aplicación de la entrada x y tan solo y como respuesta a la segunda entrada. En tal caso, dos fallos simultáneos se enmascararían mutuamente.

3.2. Procesos de pruebas basados en modelos

Los trabajos para definir procesos de pruebas basados en modelos para arquitecturas

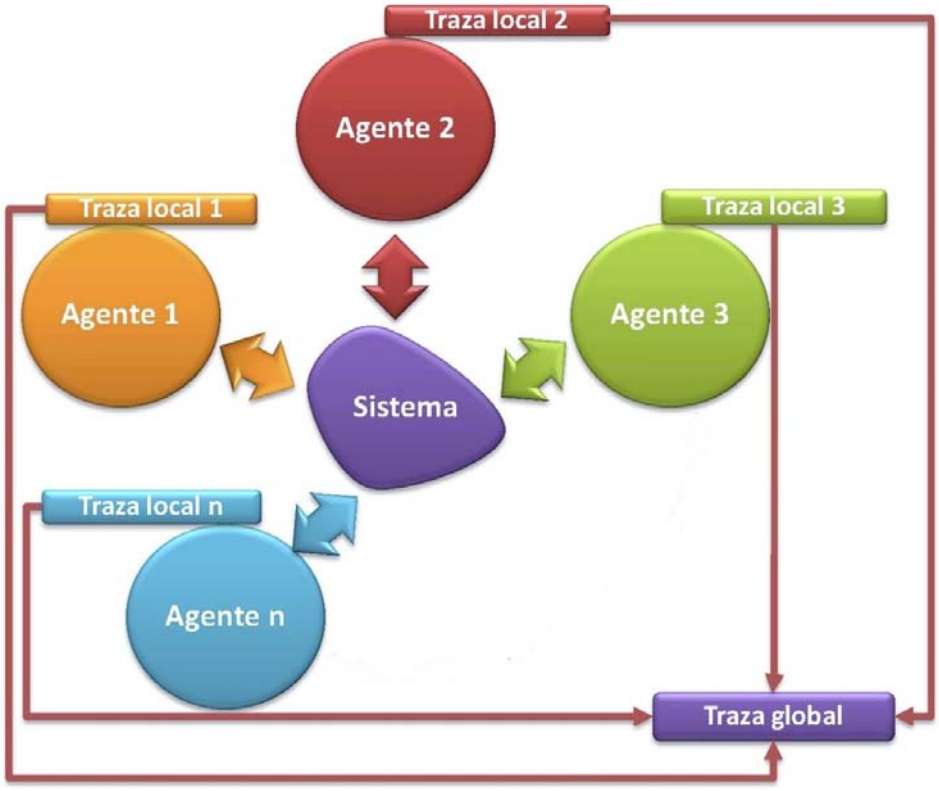

Figura 2. Esquema de la arquitectura para procesos de pruebas en sistemas altamente distribuidos.

> “Los procesos de pruebas *pasivos* parten del hecho de que no se tiene acceso directo al sistema que estamos analizando”

distribuidas se han centrado en sistemas deterministas, usualmente utilizando máquinas de estados finitos deterministas, lo que permite detectar la posible presencia de estos dos problemas en la fase de aplicación de los casos de prueba.

Aunque los formalismos deterministas suelen ser apropiados para modelar sistemas sin componentes, los sistemas distribuidos son frecuentemente no deterministas y por lo tanto estos marcos son poco adecuados. Ello ha llevado al estudio de procesos de pruebas basados en modelos para arquitecturas distribuidas sin asumir el determinismo de los sistemas considerados [7]. En este trabajo se han definido diferentes relaciones de implementación considerando dos posibles escenarios.

El primero de ellos asume que los agentes que llevan a cabo los procesos de pruebas son independientes y tan solo se requiere la consistencia de los comportamientos locales con alguno de los comportamientos establecidos en el sistema global para el puerto correspondiente.

En el segundo caso, existe la posibilidad de que el comportamiento observado en los diferentes puertos pueda analizarse conjuntamente, lo que da lugar a una relación de implementación diferente.

En este trabajo también se proporciona un método para obtener casos de prueba locales, uno para cada puerto, mediante la proyección de un caso de prueba global. Por otra parte, a pesar de que en el caso general el problema de la controlabilidad no es evitable, con el objetivo de reducirlo se restringe el marco a considerar los llamados *casos de prueba controlables*, es decir, casos de prueba que no puede generar una situación en la que el agente local tenga que o aplicar una entrada o esperar una salida, teniendo en cuenta lo que haya ocurrido en otro puerto. Además de caracterizar formalmente los casos de prueba controlables, se presenta un algoritmo para la generación de los mismos a partir de una especificación del sistema que se pretende implementar.

Como continuación de este trabajo se ha desarrollado un nuevo marco [8] para establecer de una forma más precisa la relación causal entre los eventos observados en los diferentes puertos. Para ello se han etiquetado las acciones con información temporal referente al instante en el que se ha producido la observación de las mismas, hecho que permite, en teoría, determinar el orden en el que se han producido las acciones.

Existen dos opciones alternativas. La primera asume la existencia de un reloj global. Al ser esta hipótesis poco realista, la segunda opción considera que en cada puerto se dispone de un reloj local. Si no se dispone de información referente al nivel de sincronización de dichos relojes locales, la información temporal no sería de ninguna utilidad a la hora de ordenar los eventos.

En nuestra propuesta hemos considerado que era asumible cuantificar las diferencias potenciales de tiempos entre los diferentes relojes locales y se han investigado diferentes escenarios que han dado lugar a diferentes relaciones de implementación.

Inicialmente se ha considerado una sincronización perfecta de los mismos, pero dado que esta hipótesis es de nuevo poco realista, y equivale a disponer de un reloj global, se han considerado otras alternativas.

Por una parte se ha asumido que se conoce un límite superior de la posible discrepancia de los relojes, así como que dicha diferencia puede sufrir un incremento lineal. Otra opción que se ha contemplado es que se disponga de una función h que regula dicho crecimiento, de modo que en un tiempo t la diferencia máxima entre los relojes sea $h(t)$. Finalmente, se han estudiado las relaciones entre las diferentes nociones de conformidad propuestas.

4. Procesos de pruebas pasivos

En los marcos de pruebas habituales, como los descritos en las secciones anteriores, se puede interaccionar con el sistema que se está analizando, es decir, es posible aplicar una batería de casos de prueba y observar los resultados. En contraste con este tipo de marcos, podemos mencionar los procesos de pruebas *pasivos* que, aunque con otras denominaciones, son técnicas que aparecen documentadas en la literatura al menos desde finales de los años setenta.

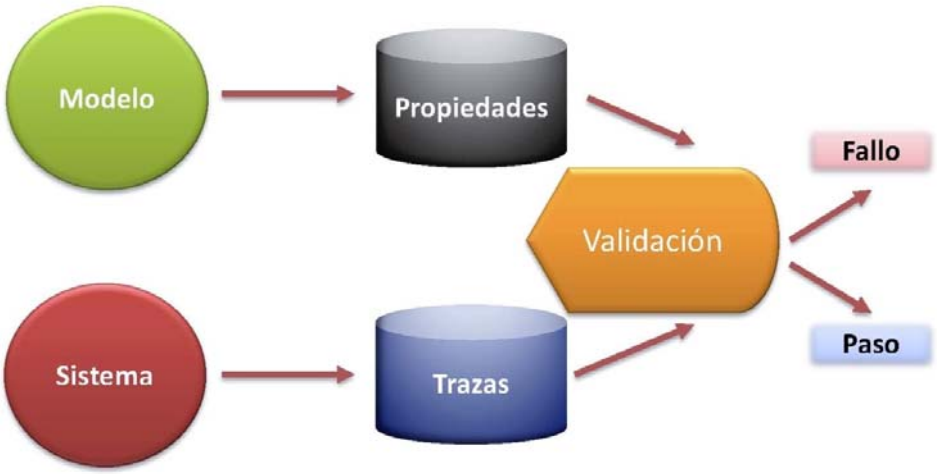

Figura 3. Esquema del proceso de pruebas pasivo.

Estos procesos parten del hecho de que no se tiene acceso directo al sistema que estamos analizando. Esta situación puede darse, por ejemplo, en grandes bases de datos donde se tiene que verificar el buen comportamiento de una funcionalidad y, a diferencia de los procesos de pruebas activos, no se permite la interacción directa con el sistema. Ello implica que no se puedan realizar operaciones como añadir datos, borrar datos, crear tablas, etc.

En este caso, se tiene que estimar la corrección del sistema a partir de la observación de las interacciones realizadas por dicho sistema con un conjunto de datos externos, comprobando si éstas son correctas con respecto a un conjunto de propiedades extraído de la especificación. En otras palabras, un proceso de pruebas pasivo consiste en enfrentar trazas observadas en el sistema con propiedades que dicho sistema debe cumplir en todo momento.

Un ejemplo típico de propiedades que deben verificarse en un sistema es:

Si detectamos un usuario que solicita conectarse al sistema y el sistema le proporcione acceso, entonces si después de realizar una serie de operaciones el usuario solicita la desconexión, ésta debe ser confirmada por el sistema.

En la **figura 3** se muestra un esquema en el que aparecen las principales componentes de un proceso de pruebas pasivo.

En [9] se presenta un marco integrado para realizar un proceso de pruebas pasivo en sistemas con información temporal. En primer lugar, se adaptan las propiedades que se pueden representar de forma que se tengan en cuenta restricciones temporales. Por ejemplo, en el ejemplo anterior podríamos asegurarnos de que el sistema siempre da acceso en menos de cinco segundos.

Un problema importante a la hora de realizar un proceso de pruebas en general es que el número de casos de prueba suele ser astronómico y es necesario establecer un criterio para seleccionar aquellos casos de prueba más relevantes.

En el caso de los procesos pasivos tenemos una situación similar. Por lo tanto, es necesario establecer un mecanismo de selección para determinar que propiedades debemos verificar en primer lugar.

Utilizando un marco basado en *mutación*, hemos analizado distintos tipos de sistemas y de propiedades y nuestras principales conclusiones han sido dos. En primer lugar, las propiedades que revisan pocas acciones del sistema tienden a ser más efectivas a la hora de encontrar acciones erróneas, pero no disponen de la capacidad de encontrar errores por cambios de estado indebidos. En segundo lugar, observamos que el número de fallos detectados crece de forma logarítmica con respecto a la longitud de las trazas observadas.

5. Conclusiones y trabajo futuro

En este artículo hemos revisado algunas de nuestras contribuciones recientes en la extensa área de trabajo conocida como *procesos de pruebas basados en modelos*. El área compagina de manera adecuada metodologías rigurosas, con una fuerte componente formal y matemática, y una visión aplicada, teniendo siempre en cuenta los sistemas reales en los que dichas metodologías se pueden aplicar.

A pesar de que continuamos trabajando en marcos teóricos que extiendan los resultados obtenidos con anterioridad, en la actualidad nuestro principal interés consiste en poner en práctica nuestras metodologías. Nuestros primeros experimentos han sido bastante exitosos y queremos aplicar nuestras propuestas en sistemas más interesantes.

En lo que respecta a los procesos de pruebas para sistemas temporizados, hemos detectado que un enfoque formal permite encontrar errores que no habían sido detectados con anterioridad.

En esta línea, nuestras metodologías son especialmente útiles en pequeños sistemas para los que se ha asegurado su fiabilidad a nivel de las acciones que realizan pero donde los aspectos temporales no se han estudiado con un marco temporizado. Entre este tipo de sistemas podemos destacar electrodomésticos y componentes de sistemas complejos, y donde el factor tiempo es crítico, como es el caso de automóviles.

En lo concerniente a los procesos de pruebas para sistemas altamente distribuidos, además de trabajar en arquitecturas *clásicas* estamos empezando a usar nuestras metodologías en sistemas *cloud*. Para estos sistemas, no solo estudiamos su comportamiento a la hora de analizar las acciones que realizan; nuestros marcos temporizados permiten estudiar propiedades no funcionales relacionadas con aspectos como el rendimiento, análisis coste/beneficio y duración de componentes.

Finalmente, los procesos de pruebas pasivos están resultando útiles para analizar la corrección de sistemas de una forma poco intrusiva con respecto a comportamientos difícilmente reproducibles y con características no funcionales, como es el caso de la seguridad en sistemas críticos.

Reconocimientos

El trabajo presentado en este artículo se ha realizado en el marco de los Proyectos TESIS y ESTuDIo (TIN2009-14312-C02-01 y TIN2012-36812-C02-01).

Referencias

[1] W. Grieskamp, N. Kicillof, K. Stobie, V. A. Braberman. "Model-based quality assurance of protocol documentation: tools and methodology". *Software Testing, Verification & Reliability, vol. 21*, nº 1, pp. 55-71, 2011.

[2] R. M. Hierons et al. "Using formal specifications to support testing". *ACM Computing Surveys, vol. 41*, nº 2, 2009.

[3] M. G. Merayo, M. Núñez, I. Rodríguez. "Formal testing from timed finite state machines". *Computer Networks, vol. 52*, nº 2, pp. 432-460, 2008.

[4] A. Núñez, M. G. Merayo, R. M. Hierons, M. Núñez. "Using genetic algorithms to generate test sequences for complex timed Systems". *Soft Computing, vol. 17*, nº 2, pp. 301-315, 2013.

[5] M. G. Merayo, M. Núñez, I. Rodríguez. "A formal framework to test soft and hard deadlines in timed Systems". *Software Testing, Verification & Reliability, vol. 22*, nº 8, pp. 583-608, 2012.

[6] M. G. Merayo, M. Núñez, I. Rodríguez. "Extending EFSMs to specify and test timed systems with action durations and time-outs". *IEEE Transactions on Computers, vol. 57*, nº 6, pp. 835-844, 2008.

[7] R. M. Hierons, M. G. Merayo, M. Núñez. "Implementation relations and test generation for systems with distributed interfaces". *Distributed Computing, vol. 25*, nº 1, pp. 35-62, 2012.

[8] R. M. Hierons, M. G. Merayo, M. Núñez. "Using Time to Add Order to Distributed Testing". 18th International Symposium on Formal Methods (FM 2012), *Lecture Notes in Computer Science vol. 7436*, 2012.

[9] C. Andrés, M. G. Merayo, M. Núñez. "Formal passive testing of timed systems: theory and Tools". *Software Testing, Verification & Reliability, vol. 22*, nº 6, pp. 365-405, 2012.

Cobertura de consultas SQL y sus aplicaciones

Javier Tuya, Claudio de la Riva, María José Suárez-Cabal, Raquel Blanco
Departamento de Informática , Universidad de Oviedo

<{tuya, claudio, cabal, rblanco}@uniovi.es>

Resumen: *El concepto de cobertura es bien conocido entre los profesionales de las pruebas del software. Asimismo, existen multitud de herramientas para medir la cobertura de las pruebas respecto de los requisitos y también del código fuente. Sin embargo, cuando se trata de la cobertura del código fuente, no se contempla la cobertura de las sentencias SQL. En este artículo se presenta el concepto de cobertura para consultas SQL y cómo se puede evaluar. Se muestran además una serie de aplicaciones de la cobertura como son la generación de bases de datos de prueba y la reducción del tamaño de bases de datos preservando la cobertura, así como un conjunto de herramientas (Test4Data) que automatizan todo lo anterior.*

Palabras clave: *Pruebas de aplicaciones con bases de datos, consultas SQL, cobertura de pruebas, herramientas de pruebas.*

Autores

Pablo Javier Tuya González es Catedrático de Universidad y director del Grupo de Investigación en Ingeniería del Software (GIIS, <http://giis.uniovi.es/>) de la Universidad de Oviedo. Actualmente es director de la Cátedra Indra-Uniovi y coordinador del grupo de trabajo de AENOR AEN/CTN 71/SC7/GT26 Pruebas del Software. Sus líneas de investigación se centran en las pruebas del software, en especial para aplicaciones con bases de datos y orientadas a servicios.

Claudio de la Riva Álvarez es Profesor Contratado Doctor del Departamento de Informática y miembro del Grupo de Investigación en Ingeniería del Software (GIIS; <http://giis.uniovi.es/) de la Universidad de Oviedo. Sus líneas de investigación se centran en las pruebas del software, en especial sobre aplicaciones que acceden a bases de datos y datos XML. Ha participado en varios proyectos de investigación relacionados con la prueba del software, tanto con financiación pública como privada.

María José Suárez-Cabal es Profesora Contratada Doctor y miembro del Grupo de Investigación en Ingeniería del Software de la Universidad de Oviedo. Su línea de investigación está orientada hacia las pruebas del software, y más en concreto sobre las aplicaciones que acceden a bases de datos. Asimismo colabora como revisora en diferentes revistas y conferencias relacionadas con la Ingeniería del Software.

Raquel Blanco Aguirre es Profesora Ayudante Doctor y miembro del Grupo de Investigación en Ingeniería del Software (GIIS) de la Universidad de Oviedo. Su línea de investigación se centra en las pruebas del software, especialmente en las pruebas de aplicaciones que acceden a bases de datos y la interacción del usuario con dichas bases de datos.

1. Introducción

Las aplicaciones que utilizan bases de datos (BD) deben gestionar grandes cantidades de información almacenada y organizada en múltiples tablas. Aunque existen sistemas de persistencia y nuevos paradigmas de bases de datos (por ej. NoSQL [1]), los sistemas de gestión de bases de datos relacionales a cuyos datos se accede utilizando consultas SQL [2] siguen siendo omnipresentes hoy en día.

Por otra parte, SQL es un lenguaje semi-declarativo que puede incluir complejos procesamientos en una simple consulta. Una consulta SELECT puede combinar datos de múltiples tablas, seleccionar aquellos que cumplen determinadas condiciones, agrupar datos de acuerdo con determinados criterios y realizar una última selección y ordenación. Adicionalmente, la lógica tri-valuada [3] por la que se rige la evaluación de valores nulos añade complicaciones semánticas adicionales.

Las técnicas utilizadas para el diseño de casos de prueba están ligadas al concepto de criterio de cobertura (de sentencias, condiciones, caminos, etc.). Mediante la aplicación de una técnica se determinan un conjunto de situaciones que deben ser probadas y, a partir de ellas, los casos de prueba definirán las condiciones iniciales, entradas y salidas deseadas para que se cubran todas las situaciones antes definidas. Se dice entonces que los casos de prueba tienen una cobertura del 100% respecto de las situaciones definidas o dicho de otra forma, que cumplen el criterio de cobertura.

La prueba de aplicaciones que acceden a bases de datos requiere preparar cuidadosamente la entrada (que es la base de datos y que por tanto puede incluir muchas tablas) y comprobar la salida (que también tiene una estructura similar a una tabla). Cuando una aplicación utiliza SQL embebido en código procedural, los conceptos convencionales de cobertura no son aplicables ya que la consulta es una línea más del código fuente y por tanto será cubierta simplemente por el hecho de ejecutarse alguna vez. Sin embargo, debido al procesamiento complejo realizado por la consulta pueden existir gran variedad situaciones que deben ser probadas, por lo que sería preciso disponer de algún tipo de definición de cobertura específicamente diseñado para SQL.

La evaluación de la cobertura de las consultas SQL de acuerdo con un determinado criterio puede ser potencialmente útil en diferentes escenarios. En primer lugar, para evaluar el grado de exhaustividad de un conjunto de casos de prueba a partir de la medición de la cobertura. También se puede emplear durante el desarrollo, donde una pequeña base de datos de prueba es creada desde cero para cumplir el criterio de cobertura definido.

Sin embargo, crear bases de datos desde cero puede ser costoso (sobre todo cuando la base de datos tiene muchas relaciones) por lo que otro escenario es utilizar una base de datos previamente poblada y completarla con datos significativos para probar la consulta. Mantener la base de datos de prueba lo más reducida posible facilita la comprobación de los resultados deseados con los obtenidos y la elaboración de pruebas adicionales, por lo que otro uso puede ser la reducción de su tamaño para obtener solamente los datos que permiten cumplir el criterio de cobertura.

En este artículo se presentan dos criterios de cobertura (uno basado en decisiones lógicas y otro basado en mutación) específicamente diseñados para tratar las particularidades del código SQL (**secciones 3 y 4**). Se muestra a continuación cómo se pueden utilizar para generar automáticamente una base de datos de prueba (**sección 5**) y para reducir la base de datos de prueba partiendo de una preexis-

> “Mantener la base de datos de prueba lo más reducida posible facilita la comprobación de los resultados deseados con los obtenidos y la elaboración de pruebas adicionales”

tente (**sección 6**). Una aplicación adicional determina la cobertura de pruebas realizadas utilizando una interfaz de usuario, donde el código de la aplicación es considerado como una caja negra (**sección 7**). Finalmente se muestra un conjunto de herramientas que automatizan todo lo anterior (**sección 8**) denominado Test4Data.

2. Concepto de cobertura para SQL

Supongamos una base de datos que contiene productos, donde cada uno de ellos puede estar asociado a un tipo de descuento. Una consulta que muestra los tipos de descuento junto con los productos a los que se aplica, tales que el tipo de descuento es superior a un valor dado (parámetro representado por $) podría tener la forma:
(Q1) `SELECT T.idTD, T.Porcentaje, P.IdProd FROM TipoDescuento T INNER JOIN Producto P ON P.idTD=T.idTD WHERE T.Porcentaje>$`

Aplicando técnicas convencionales (por ej. cobertura de decisiones), se podrían determinar dos situaciones a probar derivadas de la cláusula `WHERE`:
1) el `WHERE` es cierto.
2) el `WHERE` es falso.

Para utilizar valores límite, y puesto que además del parámetro `$` la base de datos es una entrada para la consulta, bastaría tener una base de datos de prueba con dos parejas de TipoDescuento y Producto con porcentajes 11 y 10 respectivamente (tomando el parámetro el valor 10). De esta forma la salida de la consulta mostraría el primer par y filtraría el segundo.

Este caso de prueba ejercita las dos situaciones definidas para el `WHERE` y por tanto la cobertura de la base de datos respecto de la consulta es el 100%.

Pero, ¿Es suficientemente completa esta prueba? ¿Existen otras situaciones que deberían ser probadas?

La respuesta es que esta prueba no es completa puesto que existen otras situaciones que deberían ser probadas con el objetivo de detectar potenciales defectos en la construcción de la consulta, por ejemplo:
3) Probar con filas TipoDescuento que no tienen ningún Producto relacionado. Esta situación debe ser ejercitada por la prueba puesto que si este tipo de filas de TipoDescuento debe mostrarse en la salida, la consulta es incorrecta dado que debería incluir una cláusula `LEFT JOIN` en vez de `INNER JOIN`.

4) Probar con filas TipoDescuento que tengan valor `NULL` si lo permite el esquema. Estos valores significan algo no determinado, indefinido, o no aplicable. La consulta no muestra este tipo de filas porque la condición del `WHERE` las filtra. Si tuvieran que mostrarse, la condición debería añadir "`OR T.Porcentaje IS NULL`".

En la **figura 1** se muestra una base de datos de prueba que cubre todas las situaciones anteriormente descritas (considerando que el parámetro `$` tiene el valor 10), y que por tanto representa un único caso de prueba que las ejercita. En ella se representa la salida obtenida con la consulta inicialmente descrita (Q1) y la que se obtendría con la consulta cuya especificación contempla los comportamientos definidos anteriormente (Q2). Se puede apreciar que con estos datos de entrada, a través de la comparación de las salidas se podría determinar que Q1 es incorrecta si la correcta es Q2.

Un criterio de cobertura específico para SQL debería permitir determinar este tipo de situaciones, así como permitir evaluar su cumplimiento para una base de datos dada. Esto es lo que se muestra en la siguiente sección.

3. Cobertura lógica

El criterio de cobertura lógica es denominado SQLFpc (*SQL Full Predicate Coverage*) [4] y está basado en los principios de los criterios MCDC (*Modified Condition/ Decision Coverage*) [5].

Éstos se basan en que para probar una cláusula lógica hay que incluir situaciones en las que cambiando una sola condición, mientras se mantiene el resto sin cambiar, la salida cambia. Por ejemplo para probar "a AND b" habría que probar dos situaciones en las que "a" toma valor cierto y falso respectivamente, mientras que "b" es cierto,

Tipo Descuento

idTD	Porcentaje
21	11
22	10
23	11
24	NULL

Producto

idProd	idTD
31	21
32	22
34	24

(Consulta Q1)

```
SELECT T.idTD, T.Porcentaje, P.IdProd
FROM TipoDescuento T
INNER JOIN Producto P ON P.idTD=T.idTD
WHERE T.Porcentaje>10
```

(Consulta Q2)

```
SELECT T.idTD, T.Porcentaje, P.IdProd
FROM TipoDescuento T
LEFT JOIN Producto P ON P.idTD=T.idTD
WHERE T.Porcentaje>10
        OR T.Porcentaje IS NULL
```

Salida Consulta Q1

IdTD	Porcentaje	IdProd
21	11	31

Salida Consulta Q2

IdTD	Porcentaje	IdProd
21	11	31
23	11	NULL
24	NULL	34

Figura 1. Ejemplo de base de datos de prueba.

> “Cada regla de cobertura viene acompañada de una descripción en lenguaje natural del significado de la situación de prueba, por lo que las situaciones no cubiertas indican al ingeniero las características de los datos que debe completar para aumentar la cobertura”

así como dos situaciones en las que "b" toma valor cierto y falso respectivamente, mientras que "a" es cierto. De estas cuatro situaciones, una es repetida, luego se obtienen finalmente tres situaciones a probar. La ventaja de este criterio es que escala de forma lineal cuando aumenta la complejidad de la decisión, manteniendo una buena capacidad de detección de defectos.

Para SQL, la aplicación es relativamente directa para consultas con una sola tabla en cuanto a las condiciones `WHERE` y `HAVING`. Sin embargo, el criterio SQLFpc contiene además la forma de determinar situaciones relativas a la presencia de valores nulos en las columnas de las tablas, la estructura de las cláusulas `JOIN`, las formas de agrupar filas con `GROUP BY`, la forma en la que se evalúan las funciones agregado así como otras cláusulas como `CASE`. Más detalles se pueden encontrar en [4].

Pero el objetivo es que estas situaciones se puedan obtener automáticamente y que pueda evaluarse de forma automática la cobertura de una consulta. Para ello se introduce el concepto de regla de cobertura, que representa una situación a probar. A partir de la consulta SQL original se realizan una serie de transformaciones automáticas que tienen en cuenta la propia consulta y el esquema de la base de datos, de forma que cada situación queda representada en una regla que es otra consulta SQL.

De esta forma, en el ejemplo de la sección anterior, las cuatro situaciones identificadas (WHERE cierto y falso, TipoDescuento sin producto, y valor nulo en Porcentaje) quedan representadas por las reglas que se muestran en la **figura 2** (para el parámetro con valor 10).

La característica principal de las reglas de cobertura es la facilidad para su evaluación, ya que si se ejecutan contra la base de datos de prueba obtienen alguna fila en la salida cuando la situación que representan es cubierta, y no obtienen ninguna fila cuando la situación no es cubierta.

En [4] se muestra cómo se ha evaluado la cobertura de un conjunto de consultas complejas tomadas de un ERP de código abierto (Compiere), donde incluso en las consultas más complejas (decenas de condiciones, y tablas) se obtiene un número limitado de reglas de cobertura que se ejecutan de forma eficiente (pues las reglas son también consultas SQL).

Cada regla de cobertura viene acompañada de una descripción en lenguaje natural del significado de la situación de prueba, por lo que las situaciones no cubiertas indican al ingeniero las características de los datos que debe completar para aumentar la cobertura.

4. Cobertura de mutación

La prueba de mutación (*mutation testing*) es una técnica de prueba basada en fallos originalmente propuesta en [6] y [7].

El análisis de mutación consiste en generar un conjunto de programas alternativos denominados mutantes, donde cada uno de ellos tiene un defecto simple que consiste habitualmente en un cambio sintáctico realizado al programa original. Se denomina operador de mutación a la regla que permite transformar el programa original en el programa mutado.

SQLMutation es un sistema que permite generar mutantes para consulta SQL. Dada una consulta genera un conjunto de consultas mutadas que introducen defectos en las principales cláusulas de SQL, los operadores de las expresiones, los identificadores utilizados así como otros mutantes específicos relativos al manejo de los valores NULL. Más detalles

(1) SELECT T.idTD, T.Porcentaje, P.IdProd FROM TipoDescuento T INNER JOIN Producto P ON P.idTD = T.idTD WHERE (T.Porcentaje = 11)

(2) SELECT T.idTD, T.Porcentaje, P.IdProd FROM TipoDescuento T INNER JOIN Producto P ON P.idTD = T.idTD WHERE (T.Porcentaje = 10)

(3) SELECT T.idTD, T.Porcentaje, P.IdProd FROM TipoDescuento T LEFT JOIN Producto P ON P.idTD = T.idTD WHERE ((P.idTD IS NULL) AND (T.idTD IS NOT NULL)) AND (T.Porcentaje > 10)

(4) SELECT T.idTD, T.Porcentaje, P.IdProd FROM TipoDescuento T INNER JOIN Producto P ON P.idTD = T.idTD WHERE (T.Porcentaje IS NULL)

Figura 2. Reglas de cobertura para las situaciones identificadas en el ejemplo.

sobre el conjunto de operadores de mutación y la detección de mutantes equivalentes se pueden encontrar en [8] y [9].

El uso de los mutantes de SQL es similar al de las reglas de cobertura SQLFpc descritas anteriormente. Para cada consulta se genera el conjunto de mutantes que son ejecutados contra la base de datos. La diferencia radica en la evaluación de la cobertura (denominada "*mutation score*"). En este caso se ejecuta por un lado la consulta original y por otro la consulta mutada y se comparan las salidas.

Cuando estas salidas son diferentes el mutante es cubierto (se dice que el mutante es "matado"), ya que si la consulta contuviese el defecto representado por el mutante se podría distinguir la consulta correcta de la mutada. El porcentaje total de cobertura será el porcentaje de mutantes que han sido matados respecto del total de los mutantes.

5. Generación de bases de datos de prueba

Uno de los aspectos más costosos en la prueba de consultas SQL es la preparación de la base de datos de prueba. Aunque existen algunas herramientas que permiten poblar bases de datos [10], los datos generados no son suficientemente significativos ya que únicamente tienen en cuenta la estructura de la base de datos y no la lógica de las consultas. Para ello se deben considerar las distintas situaciones de prueba derivadas de las consultas, como pueden ser las reglas de cobertura que determine el criterio SQLFpc. El objetivo es generar de forma automática los datos de prueba de manera que cada una de las reglas sea cubierta.

Dado un conjunto de consultas SQL del que se derivan un conjunto de reglas de cobertura, el procedimiento básico consiste en modelar las reglas y las restricciones impuestas por el esquema de la base de datos como restricciones a satisfacer para posteriormente, buscar una solución que cumpla dichas restricciones, por ejemplo utilizando un "*constraint solver*".

Las restricciones correspondientes a las reglas se obtienen a partir de las condiciones presentes en sus cláusulas (JOINs, WHERE, etc.). Por ejemplo, la primera regla de la **sección 2** se modelaría como `"T.IdTD = P.IdTD AND T.Porcentaje=11"`, que indica que en la base de datos de prueba debe existir una fila en Productos (P) relacionada con otra en TipoDescuento (T) y cuyo porcentaje de descuento sea 11.

La resolución del conjunto de restricciones dará como resultado un conjunto de valores en las filas de las tablas que se utilizarán para poblar la base de datos de prueba. En [11] se detalla este procedimiento y se aplica a una aplicación de HelpDesk con 7 tablas y 230 columnas en total. Se han considerado 20 consultas de distinta complejidad para las que se generan 75 reglas, obteniendo una base de datos de prueba con 139 filas con un 86,67% de cobertura.

Este procedimiento anterior es aplicable cuando se parte de una base de datos de prueba vacía y no permite generar una base de datos cuando dos o más reglas son incompatibles entre sí. Por ello se modifica el procedimiento anterior, adoptando un enfoque incremental, más cercano al proceso seguido por el ingeniero para construir sus pruebas que se describe a continuación.

En lugar de resolver todas las reglas en un único paso, cada regla se resuelve de forma individual pero teniendo en cuenta los datos generados previamente, que se suministran también como restricciones. Si para una regla no se encuentra una solución, se marcará como tal, y se continúa con la siguiente hasta que todas hayan sido tratadas. Para aquellas que no hayan podido ser resueltas, se repite el proceso partiendo de una nueva base de datos vacía.

Como resultado se obtienen una o más bases de datos de prueba, dependiendo del número de veces que sea necesario repetir el proceso, que cubren todas las reglas. En el caso de partir de un escenario donde exista una base de datos de prueba con información y sea necesario completar con nuevas situaciones, estos datos se suministrarían al inicio del proceso como restricciones.

En la **figura 3** se muestra esquemáticamente una iteración de este procedimiento.

Este nuevo enfoque, se ha aplicado también a la aplicación HelpDesk, generando tres bases de datos con 69, 13 y 10 filas que cubren 64, 8 y 3 reglas de cobertura, respectivamente, alcanzando el 100% de cobertura. Además de obtener cobertura completa y menor número de filas, el tiempo para la generación también mejora pasando de 285 a 26 segundos.

6. Reducción de bases de datos de prueba

El problema a resolver en este caso es el siguiente: Dada una base de datos de gran tamaño (BD inicial) y un conjunto de consultas SQL que se ejecutan sobre ella, conseguir una base de datos de pequeño tamaño (BD reducida) en la que la cobertura de todas las consultas SQL sea prácticamente la misma que en la BD inicial. Esta base de datos reducida será más manejable y por lo tanto más fácil de usar en la prueba, sirviendo asimismo como punto de partida para completar la cobertura.

Recuérdese que las reglas de cobertura SQLFpc producen una salida cuando la situación representada por dicha salida es cubierta por la base de datos. Retomando el ejemplo presentado en la **sección 2** y la regla de cobertura que representa la situación que requiere que existan productos con tipo de descuento mayor que 10, esta regla mostrará tantas filas como productos cumplan este requisito. Sin embargo, para que se cumpla, basta con que haya uno solo. El procedimiento de reducción consistirá en seleccionar solamente una de las filas de salida de cada regla y determinar cuáles son las filas origen que han producido esa salida. Estas filas serán insertadas en una base de datos (inicialmente vacía) que será la BD reducida.

En el caso general, existirá un conjunto de consultas (por ej. extraídas del registro de una sesión de prueba o del funcionamiento en producción), y se generarán todas las reglas de cobertura de cada una de las consultas, eliminando duplicados y repitiendo el proceso para cada una de las reglas.

Este procedimiento básico generaría una base de datos reducida de un tamaño proporcional al número de reglas, por lo que se complementa con una estrategia para reutilizar filas.

Supóngase que se está en un momento intermedio del proceso, en el que se tiene una BD reducida que cubre un conjunto de reglas que ya han sido procesadas previamente. Al ejecutar una nueva regla contra la BD inicial y examinar cada fila de la salida, es muy posible que algunas de las filas origen ya estén en la base de datos reducida, por lo que no es necesario añadirlas. Por ello, el procedimiento de selección de una fila de la salida examina cada una de ellas para determinar el coste de añadirla a la base de datos reducida. Este coste es igual al número de filas de la BD inicial que deberían ser añadidas y que no están ya presentes en la base de datos reducida. Tras examinar todas las filas, se elige el conjunto de filas origen que supone menor coste que se insertan en la BD reducida. Más detalles sobre este proceso pueden ser consultados en [12].

Este procedimiento permite conseguir altos grados de reducción incluso con bases de datos iniciales de gran tamaño y consultas muy complejas. Como muestra, los experimentos realizados con el *benchmark* TPC-H [13] que contiene 22 consultas complejas (de las que se derivan 264 reglas) sobre bases de datos de 1Gb, 10Gb y 100Gb consigue bases de datos reducidas de alrededor de 500 filas, manteniendo la misma cobertura que cuando las reglas son ejecutadas en la BD inicial.

Otra característica interesante es que el tamaño final no es dependiente del tamaño de la

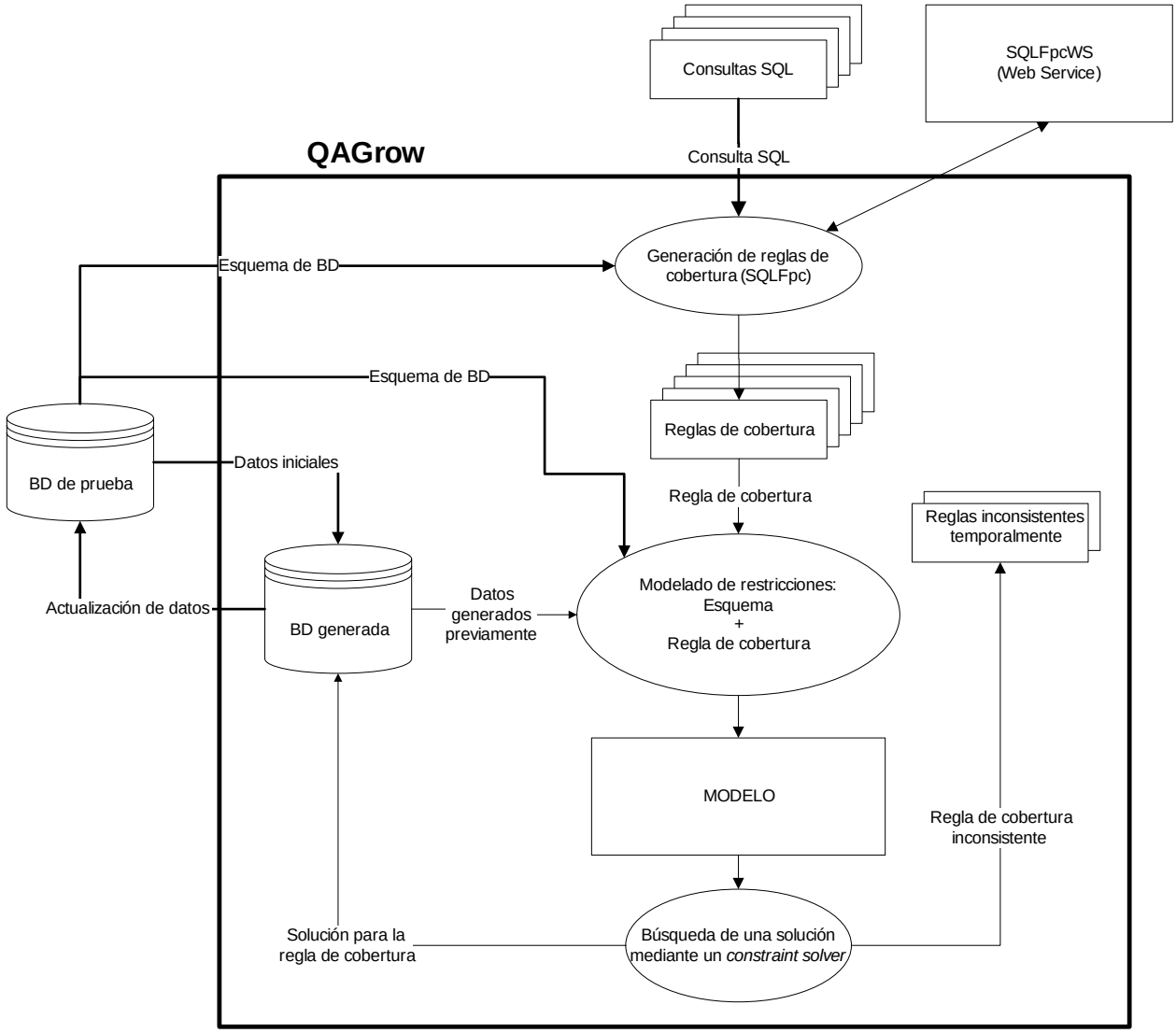

Figura 3. Generación de bases de datos prueba

base de datos original. La escalabilidad en el tiempo tiene una forma lineal, puesto que el tiempo de ejecución de todo el procedimiento de reducción es proporcional al tiempo de ejecución de las reglas de cobertura, el cual es también proporcional al tiempo de ejecución de las consultas de las cuales derivan.

7. Cobertura de las pruebas de la interfaz de usuario

Durante el proceso de prueba de una aplicación de bases de datos, no sólo se debe considerar la obtención de una base de datos de prueba sino que también se deben tener en cuenta los valores que se suministren a través de la interfaz de usuario. Por ello, los objetivos que se plantean en esta sección son la obtención de situaciones de prueba que consideren la base de datos y la interfaz de usuario, partiendo de la especificación de la aplicación en vez del código SQL, y la automatización de la evaluación de la cobertura alcanzada por los casos de prueba generados.

Para ello, en [14] se ha propuesto el modelado de la especificación de la aplicación, que incluye la descripción de funcionalidad requerida, la estructura de la base de datos y la estructura de la información manejada por el usuario (la interfaz de usuario).

Dado que la entrada del usuario y la base de datos de prueba están estrechamente relacionadas, la interfaz de usuario y la base de datos se modelan de forma conjunta en un modelo relacional denominado Modelo de Datos Integrado (IDM: "*Integrated Data Model*"), el cual contiene también la estructura de los casos de prueba diseñados.

Asimismo, la funcionalidad requerida de la aplicación se modela mediante un conjunto de reglas de negocio que definen las propiedades que deben cumplir los datos modelados por el IDM, las acciones que se pueden realizar sobre dichos datos y las salidas a dichas acciones. Estas reglas forman un modelo denominado Modelo de Reglas Integrado (IRM: "*Integrated Rule Model*"), a partir del cual se derivan las situaciones de prueba utilizando un criterio basado en MCDC.

Para automatizar el proceso de derivación de las situaciones de prueba, las reglas del IRM se transforman en consultas SQL y sobre ellas se aplica el criterio SQLFpc. El conjunto de reglas de cobertura derivado es posteriormente procesado (modificando y eliminando algunas reglas y añadiendo otras nuevas) para obtener las situaciones de prueba que se adecúan a la aplicación del criterio MCDC sobre las reglas del IRM.

La evaluación automática de las situaciones de prueba se consigue ejecutando las reglas de cobertura obtenidas sobre una base de datos de prueba que representa al IDM, en la cual el ingeniero introducirá los casos de prueba necesarios para cubrir dichas reglas. La **figura 4** muestra de forma esquemática el procedimiento seguido para la obtención de

> “Durante el proceso de prueba de una aplicación de bases de datos, no sólo se debe considerar la obtención de una base de datos de prueba sino que también se deben tener en cuenta los valores que se suministren a través de la interfaz de usuario”

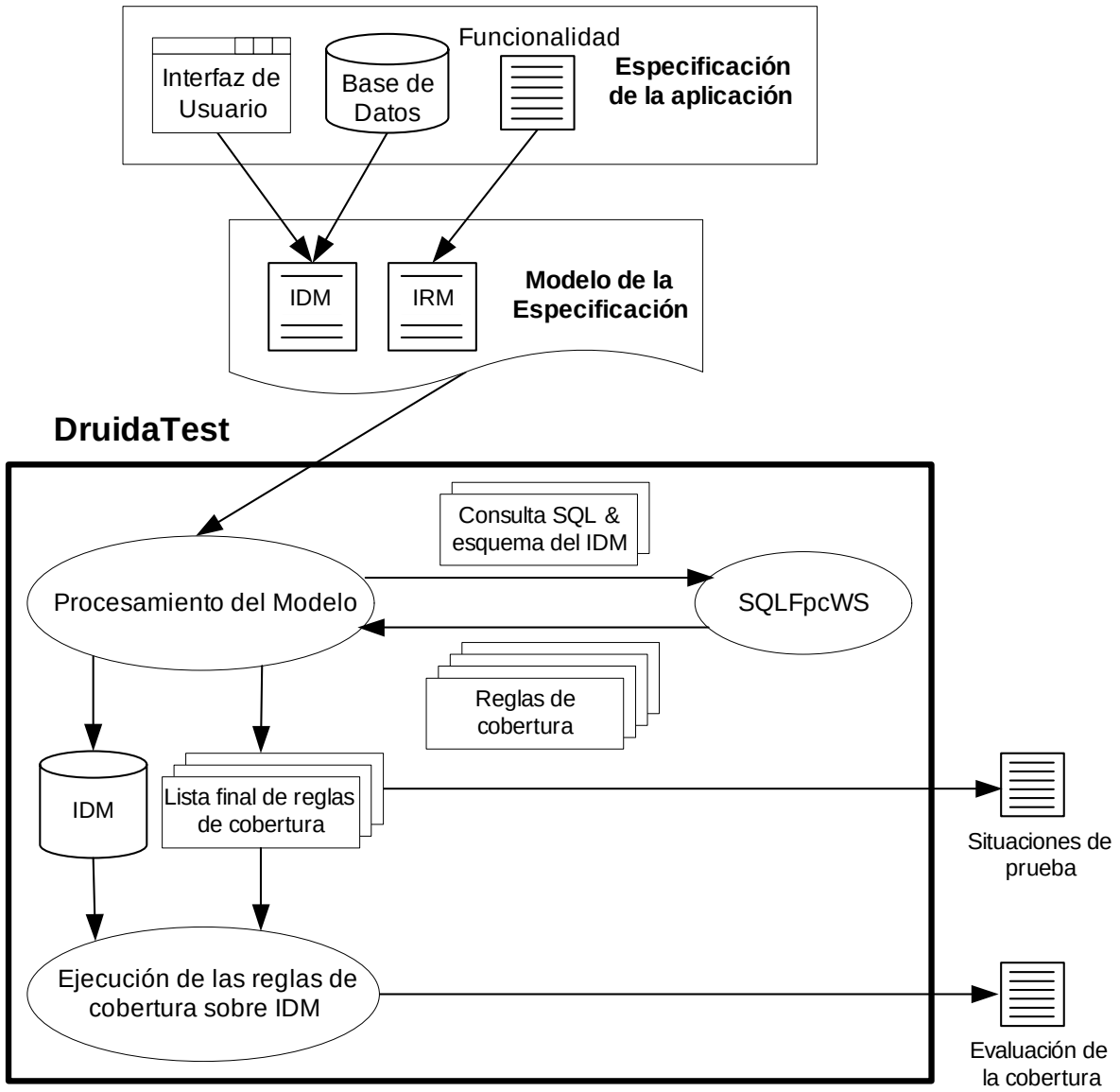

Figura 4. Cobertura de las pruebas de la interfaz de usuario.

las situaciones de prueba y la evaluación de la cobertura.

En [15] se presentan dos casos de estudio que muestran que el procedimiento permite guiar la generación de casos de prueba que detectan fallos importantes.

En una implementación del *benchmark* TPC-C denominada *BenchmarkSQL* se detectaron 24 fallos utilizando 41 casos de prueba, mientras que en la aplicación de facturación FACT, de los 22 fallos inyectados en la misma, se detectaron 21 utilizando 8 casos de prueba.

8. Soporte automatizado

Test4Data <http://giis.uniovi.es/tools/> es un *framework* de herramientas que automatizan lo descrito anteriormente. Su arquitectura se muestra en la **figura 4**.

Sus componentes principales son:

- **SQLFpc**: Permite la generación de las reglas de cobertura para el criterio del mismo nombre desde una interfaz web. Se suministra también un servicio web (SQLFpcWS) para su integración con otras herramientas.
- **SQLMutation**: Permite la generación de mutantes de SQL desde una interfaz web. Se suministra también un servicio web (SQLMutationWS) para su integración con otras herramientas.
- **SQLRules**: Permite generar reglas de cobertura y mutantes, así como la evaluación de la cobertura obtenida contra una base de datos.
- **QAShrink**: Crea una base de datos reducida partiendo de otra base de datos y un

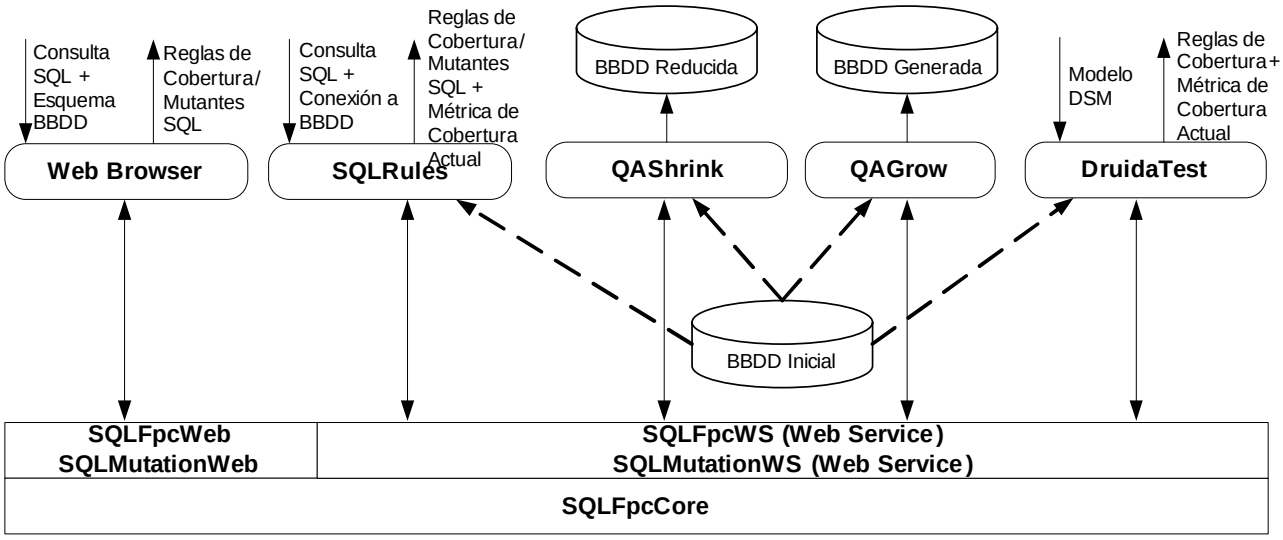

Figura 5. Arquitectura de Test4Data.

conjunto de sentencias SQL.

- **QAGrow**: Partiendo de un conjunto de consultas SQL, genera (o completa) de forma automática una base de datos de prueba para dichas consultas.
- **DruidaTest**: Es un prototipo para la evaluación de la cobertura de las pruebas realizadas desde una interfaz de usuario.

9. Conclusiones

En este artículo se han presentado dos conceptos de cobertura para consultas SQL y varias de sus aplicaciones que permiten generar bases de datos de pruebas y reducir el tamaño de bases de datos de forma que se preserve la cobertura de las consultas.

Esto permite tanto al ingeniero de pruebas como al desarrollador la evaluación de la calidad de las pruebas a partir de la evaluación de la cobertura. Facilita asimismo la tarea de diseño de las bases de datos de pruebas, manteniendo éstas con un tamaño reducido y con datos muy significativos para ejercitar la lógica incluida en las consultas SQL. Adicionalmente, se facilita la ejecución de las pruebas en lo relacionado con la carga de las bases de datos de prueba y la comparación de los resultados de las pruebas.

Agradecimientos

Este trabajo ha sido financiado por el Ministerio de Economía y Competitividad y fondos FEDER en el marco del Plan Nacional de I+D+i, proyectos TIN2007-67843-C06-01, TIN2010-20057-C03-01.

Referencias

[1] A.B.M. Moniruzzaman, S.A. Hossain. NoSQL Database: New Era of Databases for Big data Analytics - Classification, Characteristics and Comparison. *International Journal of Database Theory and Application*, 6(4), 2013.

[2] ISO/IEC 9075 - International Standards Organisation. *ISO/IEC 9075, Information technology - Database languages – SQL*, 1999.

[3] N. Waraporn, K. Porkaew. Null semantics for subqueries and atomic predicates. *IAENG International Journal of Computer Science* 35 (3), 2008.

[4] J. Tuya, M. J. Suárez-Cabal, C. de la Riva. Full predicate coverage for testing SQL database queries. *Software Testing, Verification and Reliability*, 20(3), pp. 237-288, September 2010.

[5] J.J. Chilenski. An investigation of three forms of the modified condition decision coverage (MCDC) criterion. *Technical Report DOT/FAA/AR-01/18*, U.S. Department of Transportation, Federal Aviation Administration, April 2001.

[6] R.A. DeMillo, R.J. Lipton, F.G. Sayward. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11(4), pp. 34-43, 1978.

[7] R.G. Hamlet. Testing programs with the aid of a compiler. *IEEE Transactions on Software Engineering* 3(4), pp. 270-290, 1977.

[8] J. Tuya, M. J. Suarez-Cabal, C. de la Riva. SQLMutation: a Tool to Generate Mutants of SQL Database Queries. *Second Workshop on Mutation Analysis (Mutation2006)*. Raleigh, North Carolina, November 2006.

[9] J. Tuya, M. J. Suárez-Cabal, C. de la Riva. Mutating database queries. *Information and Software Technology*, 49(4), pp. 398-417, April 2007.

[10] N. Bruno, S. Chaudhuri. Flexible database generators. *31st International Conference on Very Large Data Bases (VLDB)*, pp. 1097-1107, 2005.

[11] C. de la Riva, M. J. Suárez-Cabal, J. Tuya. Constraint-based Test Database Generation for SQL Queries. *5th International Workshop on Automation of Software Test (AST)*. Cape Town, South Africa, May 2010.

[12] J. Tuya, M. J. Suarez-Cabal, C. de la Riva. Query-Aware Shrinking Test Databases. Second *International Workshop on Testing Database Systems (DBTest)*. Providence, RI, June 2009.

[13] TPC 2013 - Transaction Processing Performance Council. *TPC Benchmark™H (TPC-H)*. <http://www.tpc.org/tpch/> (último acceso: 10 de junio de 2013).

[14] R. Blanco, J. Tuya, R. V. Seco. "Test adequacy evaluation for the user-database interaction: a specification-based approach", *5th International Conference on Software Testing, Verification and Validation (ICST 2012)*, IEEE Computer Society, abril 2012, pp. 71-80.

[15] R. Blanco, J. Tuya, R. V. Seco. "Evaluación de la cobertura en la interacción usuario-base de datos utilizando un enfoque de caja negra", *XVII Jornadas de Ingeniería del Software y Bases de Datos (JISBD 2012)*, septiembre 2012, pp. 329-342.

Javier Ferrer, Francisco Chicano, Enrique Alba
Universidad de Málaga, España

<{ferrer, chicano, eat}@lcc.uma.es>

Algoritmos bio-inspirados para la automatización de pruebas de software en la industria

Resumen: *El software se puede encontrar en todo tipo de dispositivos electrónicos que se entremezclan en nuestro día a día. La principal manera de asegurar su calidad es mediante la realización de diferentes pruebas del software durante y después de su desarrollo. En este trabajo vamos a estudiar las pruebas unitarias, las pruebas de regresión, las pruebas de interacción y las secuencias de pruebas. Todas las pruebas realizadas en un proyecto suponen alrededor de un 50% del costo total, por lo que la automatización de la generación de datos de prueba supondría un ahorro enorme. Para ello, la transferencia Universidad-Empresa es fundamental para la aplicación de nuevas técnicas de optimización, como los algoritmos bio-inspirados, a problemas NP-difíciles como es el de la generación automática de casos de prueba. Además, como línea de trabajo futuro evaluada ya parcialmente, destacamos la generación de escenarios de validación de la robustez de programas de ciclos de semáforos para "ciudades inteligentes".*

Palabras clave: *Algoritmos bio-inspirados, Ingeniería del Software Basada en Búsqueda, pruebas de software.*

Autores

Francisco Javier Ferrer Urbano es Ingeniero en Informática por la Universidad de Málaga y actualmente estudiante de doctorado en el departamento de Lenguajes y Ciencias de la Computación de dicha universidad. Su tesis doctoral se centra en la generación automática de casos de prueba con ayuda de algoritmos de optimización metaheurísticos.

José Francisco Chicano García es profesor contratado doctor en el departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga. Su principal línea de investigación se centra en la aplicación de algoritmos de búsqueda y optimización a problemas definidos dentro de la Ingeniería del Software.

Enrique Alba Torres es catedrático en el departamento de Lenguajes y Ciencias de la Computación de la Universidad de Málaga. Sus intereses en investigación incluyen el estudio, diseño y aplicación de algoritmos metaheurísticos a problemas en los dominios de Telecomunicaciones, Ingeniería del Software, Ciudades Inteligentes, Redes Vehiculares y Optimización Combinatoria.

1. Introducción

Hoy en día podemos encontrar software para todo tipo de dispositivos electrónicos: en nuestros hogares, en nuestros automóviles, en nuestros centros de trabajos, incluso tenemos software muy avanzado en nuestros teléfonos móviles.

El software mueve el transporte en la ciudad, nos permite ver la televisión, nos pone en contacto con nuestra familia durante un viaje, construye materiales para las construcciones modernas y permite diseñar y fabricar instrumentos de precisión para el médico que salva la vida en un quirófano a otras personas. Todo este avance hace necesaria la mejora continua de los procesos de desarrollo y el mantenimiento de unos estándares de calidad elevados en el desarrollo de software.

Hay muchas actividades en Ingeniería del Software relacionadas con la calidad del software, incluyendo las más básicas dirigidas a asegurar que funciona correctamente como se espera. Una de las más importantes consiste en llevar a cabo distintos tipos de pruebas de los sistemas durante el desarrollo y en la fase de mantenimiento. Se estima que el 50% de los costes de un proyecto software son derivados de dicha fase de pruebas [11]; esta cifra se puede elevar aún más en el caso de software crítico, como el que controla centrales nucleares, vehículos espaciales, aviones, instrumentación médica o sistemas financieros, entre otros. Por esta razón, tanto la industria como la academia consideran las pruebas de software de gran interés.

La generación automática de casos de prueba se ha convertido en los últimos años en un tópico muy popular en el campo de la Ingeniería del Software. La automatización de la generación de casos de prueba puede contribuir a reducir notablemente los costes del proyecto, a la vez que separa el proceso de generación con el proceso de comprobación del resultado de las pruebas (*oracle problem* [1] – problema del oráculo). Sin embargo, el problema no es sencillo; baste decir como ilustración que si el objetivo es buscar un conjunto de pruebas que consiga maximizar la cobertura de ramas nos enfrentamos a un problema indecidible.

Debemos conformarnos, por lo tanto, con resolver el problema sólo en algunos casos y, probablemente, de una forma sub-óptima. Esto es especialmente cierto si hablamos de programas reales, con cientos de miles de líneas de código escritas en un lenguaje de alto nivel. Es al enfrentar sistemas reales con largos programas y grandes paquetes software, a veces creados de manera deslocalizada, cuando la elección de las técnicas para definir los casos de prueba se convierte en crítica e innovadora a la vez.

Los algoritmos de optimización bio-inspirados [9] son unos grandes aliados cuando nos enfrentamos a problemas NP-difíciles[1] como el que nos ocupa, con un espacio de búsqueda extremadamente grande y complejo (ver **figura 1**). Estos algoritmos basan su comportamiento en el de algunos seres vivos. Por ejemplo, los algoritmos evolutivos están inspirados en la teoría de la evolución de Darwin, mientras que la optimización basada en cúmulos de partículas (PSO, *Particle Swarm Optimization*) imitan las bandadas de pájaros o los bancos de peces. El resultado son algoritmos que ofrecen una solución de buena calidad a un problema, sin garantías de encontrar la mejor posible pero que son capaces de hacerlo en un tiempo razonable, siendo en muchos casos la única alternativa viable a un algoritmo exacto que requiere un tiempo abrumadoramente grande para encontrar la solución óptima.

La generación automática de casos de prueba usando algoritmos bio-inspirados se enmarca dentro de un área de investigación más general denominado Ingeniería del Software Basada en Búsqueda o SBSE por sus siglas en inglés (*Search-Based Software Engineering*), fundada por Mark Harman y Bryan F. Jones hace algo más de 10 años [10].

La idea principal en SBSE es la de modelar los problemas de Ingeniería del Software como problemas de optimización y aplicar las técnicas de este tipo para resolverlos. Además de la ya mencionada generación automática de casos de prueba (uno de los

> “Los algoritmos de optimización bio-inspirados son unos grandes aliados cuando nos enfrentamos a problemas NP-difíciles como el que nos ocupa, con un espacio de búsqueda extremadamente grande y complejo”

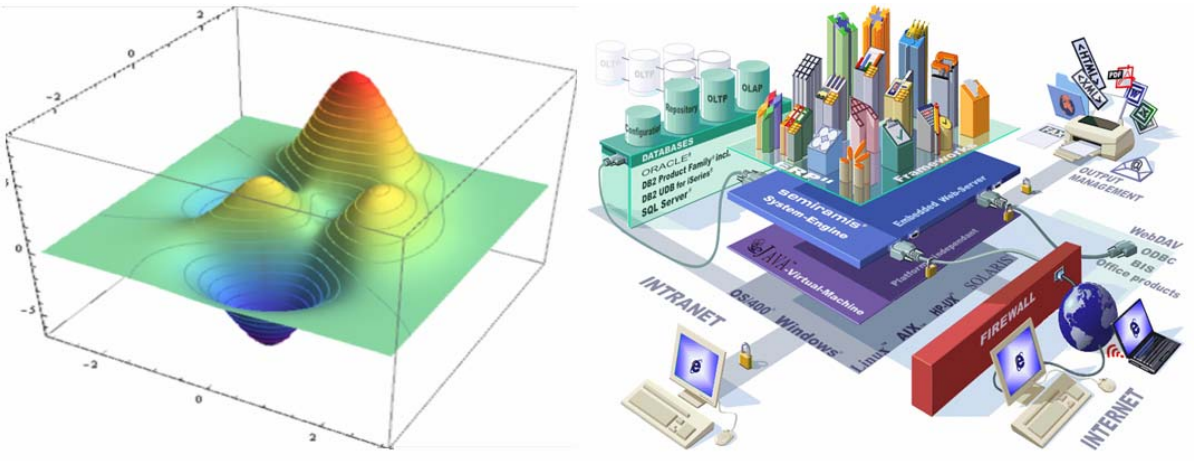

Figura 1. Problema de optimización de la automatización de pruebas interacción.

problemas que más atención concentra), otros ejemplos de problemas son la asignación de tareas y recursos en proyectos software (*Software Project Scheduling*) [2] o la selección de los requisitos que se desean implementar en la siguiente versión de un producto (*Next Release Problem*) [12] para incluir las preferencias de los usuarios y minimizar costes a la vez.

2. Desafíos científicos

La investigación en SBSE ha crecido considerablemente en los últimos años (ver **figura 2**). En esta etapa inicial de esta rama científica se han modelado para la búsqueda un gran número de problemas de Ingeniería del Software y se han resuelto con éxito usando técnicas bio-inspiradas. Este crecimiento ha tenido lugar en varias áreas, pero especialmente se concentra en el área de pruebas y depuración, como podemos ver en

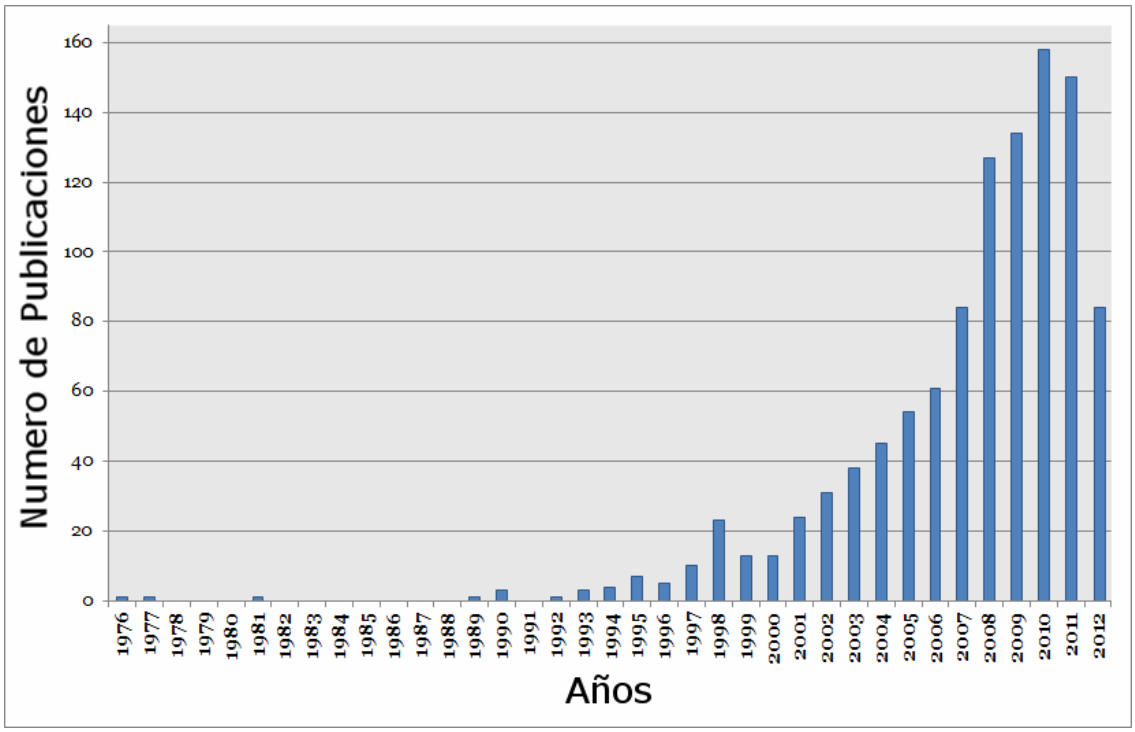

Figura 2. Número de publicaciones sobre SBSE por año (repositorio CREST/SBSE[2]).

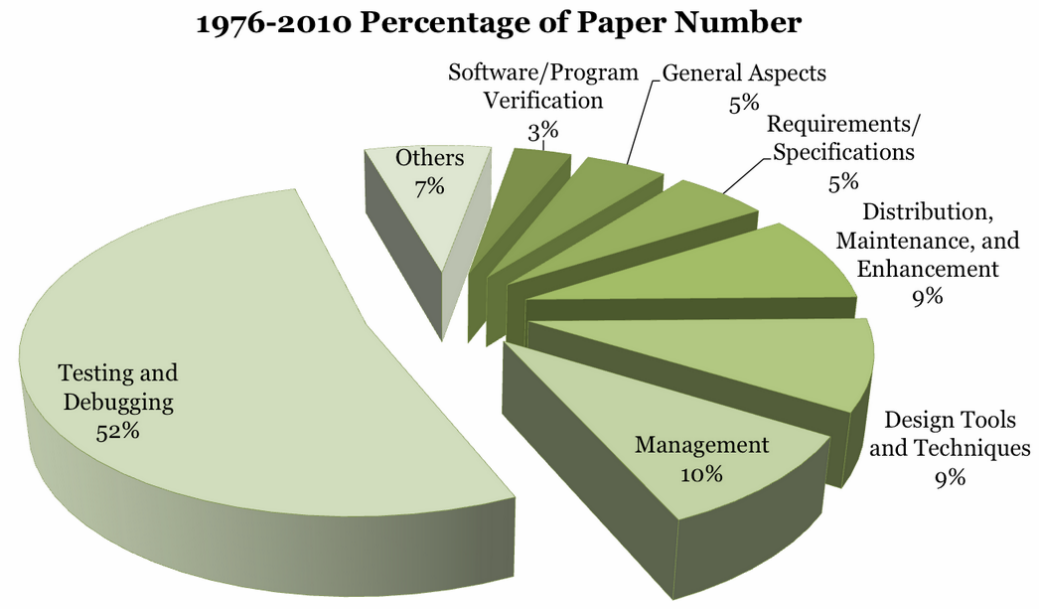

Figura 3: Ratio de publicaciones por campos en Ingeniería del Software (repositorio CREST/SBSE [2]).

la taxonomía de la **figura 3**. Por esta razón, en este trabajo vamos a centrarnos en algunos de los problemas más importantes en generación automática de casos de prueba.

En particular, primero mencionamos las áreas de trabajo de los autores de este artículo para luego describir en detalle nuestros nuevos métodos de resolución bio-inspirados (ver **figura 4**):

- **Pruebas unitarias**: son aquellas en las que se comprueba que cada unidad de código (clase, procedimiento, función,…) cumple las especificaciones. Las pruebas unitarias son la base de las pruebas de software y se encuentran en el nivel más bajo de abstracción (están ligadas al lenguaje de programación utilizado para el desarrollo).
- **Pruebas de regresión**: tienen como objetivo asegurar que no se introducen errores en las futuras versiones del producto que afecten a la funcionalidad ya implementada. Las pruebas de regresión son fundamentales para los grandes proyectos software, pues el mantenimiento de las siguientes versiones es un problema de gran magnitud.
- **Pruebas de interacción**: consisten en comprobar todas las combinaciones posibles de entornos de ejecución del software (distintos sistemas operativos, distintas configuraciones de memoria, etc.).
- **Secuencias de pruebas**: se utilizan cuando es costoso situar el sistema que se está probando en el estado requerido para la prueba. En estos casos resulta más conveniente realizar varias pruebas en secuencia para reducir costes. Por ejemplo, para probar el sistema ABS de frenado de un automóvil es necesario que dicho vehículo se encuentre a una velocidad determinada, así que es más eficiente comprobar la aceleración y después el sistema de frenado.

Figura 4. Un posible enfoque bio-inspirado de las pruebas de software.

A continuación, vamos a exponer algunos de nuestros resultados para cada uno de los tipos de prueba que hemos presentado anteriormente.

2.1. Pruebas unitarias

En este campo hemos seguido varias líneas de investigación. Por un lado, hemos tratado el problema con un enfoque multi-objetivo, en el que se debe minimizar el tamaño del conjunto de casos de pruebas y maximizar la cobertura de código a la vez.

Estamos ante un problema con dos objetivos contrapuestos que hemos resuelto usando 5 algoritmos multi-objetivo (NSGA-II, MOCELL, SPEA2, PAES, RandomSearch-multi) y 3 algoritmos mono-objetivo (GA, ES, y RandomSearch-mono) seguidos de un

post-procesamiento para minimizar el conjunto de casos de prueba [3]. Los resultados indican que el mejor de los algoritmos multi-objetivo es un algoritmo celular, MOCELL; es el más rápido y obtiene buenas soluciones con respecto al tamaño y calidad del conjunto de casos de prueba. El mejor de los algoritmos mono-objetivo, GA, es más lento por su post-procesamiento pero es capaz de alcanzar niveles de cobertura mayores que los algoritmos multi-objetivo.

En una segunda línea de investigación hemos estudiado cómo comparar el rendimiento de las herramientas de generación automática de casos de pruebas en ausencia de un *benchmark* reconocido en el estado del arte.

Para ello hemos diseñado un algoritmo que genera programas sintéticos con diversas estructuras de control, número de condiciones en dichas estructuras, diferentes grados de anidamiento, y longitud de programa [4]. El objetivo de este generador de programas es ofrecer de manera estructurada programas diferentes para que los investigadores puedan probar sus herramientas frente a multitud de escenarios.

Al realizar pruebas con programas sintéticos y programas reales, hemos comprobado que las correlaciones entre las medidas estáticas del código y la cobertura son similares [5], por lo que estos programas sintéticos parecen ser válidos para estudiar las herramientas de generación automática de casos de prueba.

Además, también nos hemos enfrentado a la generación de datos de prueba para programas orientados a objetos (OO). Una de las cuestiones críticas en este área es la selección de los objetos adecuados como entrada de los métodos que probar. La selección automática de objetos en las pruebas de programas OO [6] ha sido en parte resuelta gracias a operadores genéticos basados en la estructura jerárquica de los objetos. Así, si el objeto elegido no es correcto pero pertenece a la misma rama jerárquica lo penalizaremos menos que si pertenece a otra rama completamente diferente.

Esta investigación referente a pruebas unitarias puede repercutir beneficiosamente en las empresas que apuesten por la automatización de la generación de datos de prueba. El empleo de propuestas multi-objetivo hace más flexible la decisión sobre la *test suite*[3] que ejecutar, puesto que esta técnica deja en manos del experto la elección de la solución.

Dependiendo de los requisitos de tiempo y coste, el experto será el encargado de seleccionar la solución, ponderando cobertura o tamaño de la *test suite* según convenga.

Respecto a las pruebas orientadas a objetos, aún no se encuentran al nivel de madurez que las pruebas para los lenguajes procedimentales. Quedan algunas cuestiones por resolver asociadas a las características propias de la OO, como el polimorfismo o la vinculación dinámica. Sin embargo, la investigación avanza rápidamente y gracias a aportaciones como [6] será posible automatizar la generación de datos para programas OO con una gran eficacia.

2.2. Pruebas de regresión

Las pruebas de regresión han sido estudiadas por muchos autores en la literatura, se han desarrollado diversas técnicas para seleccionar el mínimo conjunto de pruebas que cubre todo el sistema, lo que a la postre conlleva un ahorro en tiempo debido a la eliminación de pruebas redundantes.

En este campo hemos realizado un estudio abstracto del *landscape* asociado al problema, es decir del *espacio matemático donde buscar una solución.* En particular, hemos encontrado la *descomposición en landscapes elementales* de dicho problema, gracias a la cual podemos mejorar los algoritmos de búsqueda local porque es posible predecir con cierta probabilidad si al modificar una solución vamos a estar más cerca de nuestro objetivo.

Nuestros resultados en este campo [7] indican que un algoritmo con el operador de predicción es mucho más eficiente que este mismo algoritmo con un operador de búsqueda local exhaustiva. Al igual que en una partida de ajedrez los jugadores pueden anticipar varios movimientos, nuestro operador admite varios niveles de predicción, es decir, podemos configurarlo para que explore sucesivas modificaciones de una solución, y no sólo una simple modificación sobre la solución. Esto hace que podamos regular la potencia del operador y, en consecuencia, su tiempo de ejecución.

El análisis de los espacios de búsqueda puede ser un factor diferenciador en un ámbito tan competitivo como el empresarial. Con una herramienta como la propuesta, basada en la *descomposición de landscapes elementales*, se podrán calcular *test suites* mínimas de forma eficiente, gracias al conocimiento extraído del análisis del espacio de búsqueda. Las implicaciones de estas técnicas en los grandes proyectos software pueden llegar a ser fundamentales para ahorrar tiempo/coste en las sucesivas fases de pruebas.

2.3. Pruebas de interacción

Las pruebas combinatorias de interacción permiten detectar fallos causados por la interacción de parámetros de entrada en un programa.

El ingeniero de pruebas debe identificar los aspectos relevantes de la aplicación y definir los correspondientes parámetros, con diversos valores. Estos parámetros deben ser conjuntos disjuntos. Así que un caso de prueba será un conjunto de *n* valores, uno por cada parámetro.

El problema de determinar un conjunto de casos de prueba que cubra todas las interacciones entre parámetros es un problema muy difícil. Además, en ocasiones no se tienen los recursos necesarios para ejecutar todo el conjunto de pruebas, por lo que se deben priorizar unas pruebas sobre otras.

En este dominio, los resultados de nuestro algoritmo genético priorizado [8] han sido comparados con 4 algoritmos del estado del arte usando un *benchmark* extraído de la literatura. Nuestro algoritmo ha sido el mejor en esa comparativa, que se compone de 8 escenarios con 4 distribuciones de probabilidad diferentes para asignar las prioridades. Este resultado sienta un precedente, al ser la primera vez que se usa una técnica bioinspirada para resolver este problema.

Dentro del área de las pruebas combinatorias, estamos además trabajando en las líneas de productos software (SPL, *Software Product Lines*). Una línea de productos software es un conjunto de programas relacionados que comparten varias funciones pero difieren en otras.

Las líneas de productos no son algo exclusivo de la Ingeniería del Software, Si pensamos en la industria de la telefonía móvil, una línea de producto estaría formada por las diversas versiones de un teléfono móvil, que contienen muchas características comunes pero difieren en otras como el GPS, la cámara fotográfica, o el sistema operativo.

En este caso, hemos diseñado un algoritmo genético constructivo que es capaz de generar el mínimo número de pruebas para analizar todos los pares de características de una línea de productos software. Actualmente, estamos trabajando con un algoritmo que hace uso de resolutores de satisfacibilidad[4] (SAT-*solvers*) para la generación de conjuntos de pruebas óptimos que minimizan el número de productos y maximizan la cobertura. Gracias a esta nueva aportación, podremos ejecutar el conjunto de casos de prueba que deseemos en función del tiempo disponible para su ejecución, sabiendo que es la mejor opción posible, puesto que las técnicas empleadas son exactas.

Nuestros resultados en esta área han sido beneficiosos para el sector empresarial, particularmente el trabajo realizado en [8] es una colaboración con una empresa internacional, donde tanto la empresa como los

> “En la actualidad, una de nuestras líneas de trabajo consiste en aplicar un algoritmo basado en colonias de hormigas al problema de la generación de secuencias de pruebas”

investigadores nos hemos visto beneficiados ampliamente.

En esta transferencia Universidad-Empresa hemos explorado enfoques combinatorios tanto multi-objetivo como mono-objetivo, obteniendo finalmente un algoritmo capaz de batir al algoritmo que la empresa había integrado en su herramienta profesional. Gracias a esta colaboración hemos sentado las bases para la mejora de la herramienta de la empresa, a la vez que los investigadores han tenido contacto con problemas del mundo real.

2.4. Secuencias de pruebas

En la actualidad, una de nuestras líneas de trabajo consiste en aplicar un algoritmo basado en colonias de hormigas al problema de la generación de secuencias de pruebas. Este algoritmo usa un conjunto de agentes (hormigas) que van depositando feromonas por los estados (posibles del software) que van visitando. Si una hormiga encuentra rastros de feromona avanza con más probabilidad hacia ese estado. Finalmente, la secuencia de estados que recorren las hormigas con más probabilidad es la secuencia que estamos buscando. En nuestra propuesta somos capaces de probar sistemas concurrentes usando de forma dinámica tantas hormigas como estados concurrentes tenga el sistema.

El estudio de las secuencias de pruebas es muy interesante para las empresas de software. De hecho, este trabajo también surge de las necesidades de una empresa de optimización.

Los resultados previos a esta colaboración son batidos por nuestro algoritmo de colonias de hormigas, con un beneficio claro tanto en la longitud de la secuencia como sobre la cobertura. En algunos casos complejos, el hecho de recorrer el grafo generado por los posibles estados del SUT (System Under Testing) puede suponer un desafío para los computadores actuales. Por lo tanto, creemos que las aportaciones realizadas en el campo de las secuencias de pruebas van a ser beneficiosas para el sector empresarial.

3. Transferencia a la industria

La transferencia Universidad-Empresa es un aspecto fundamental para que el progreso en las técnicas científicas se emplee en una modernización del entorno empresarial. Esto adquiere especial relevancia en el ámbito de la optimización, debido al ahorro de costes que supone la optimización de un proceso, en nuestro caso, en Ingeniería del Software.

En nuestro compromiso con la sociedad, como personal investigador, hemos realizado diversas colaboraciones con empresas nacionales e internacionales. Los beneficios de este tipo de colaboraciones son bidireccionales: las empresas se nutren de técnicas modernas para resolver los problemas de forma eficiente y, por otro lado, los investigadores públicos obtienen datos reales de las empresas para afinar sus técnicas en entornos industriales.

Hoy en día, la investigación para las empresas es un nuevo factor diferenciador en un entorno cada vez más competitivo. En esta realidad, son bastantes las empresas que han decidido colaborar con grupos de investigadores para acometer problemas que son incapaces de resolver con técnicas tradicionales.

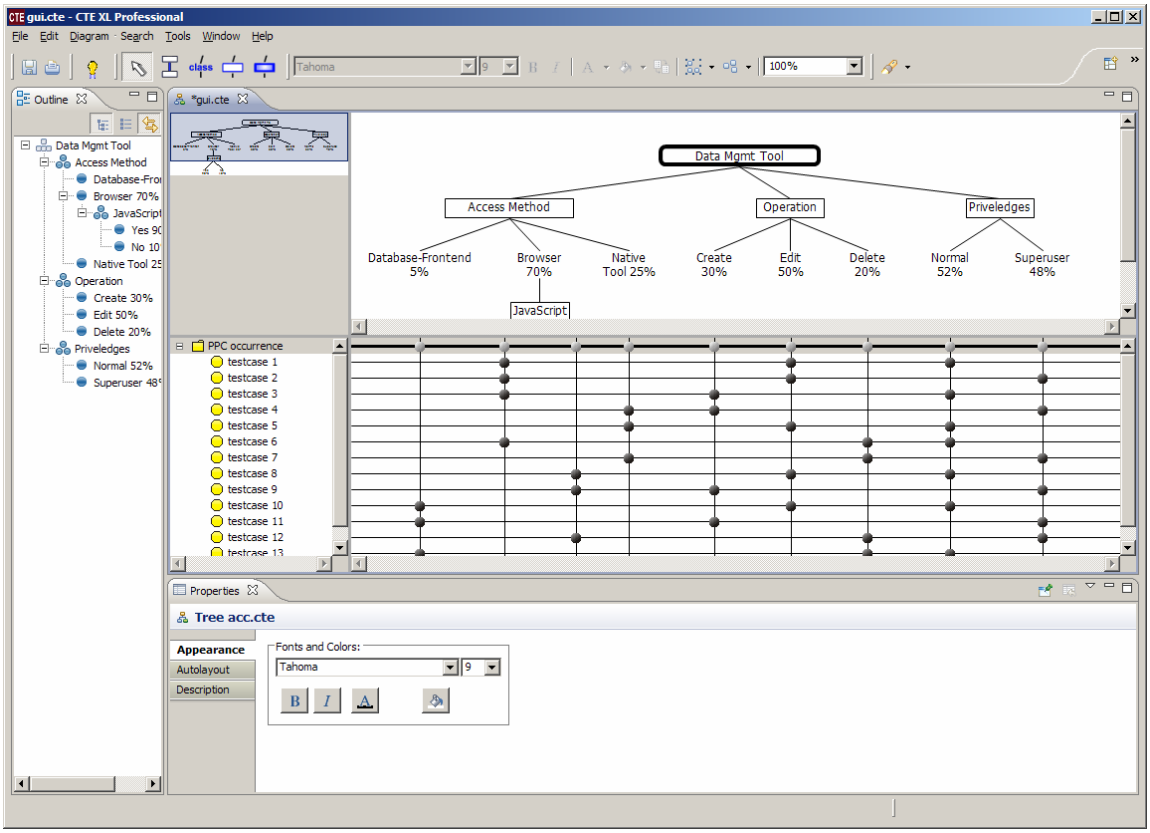

Figura 5. Herramienta CTE XL de la empresa Berner&Mattner, para el diseño de pruebas.

> De especial interés es la colaboración con la empresa alemana Berner&Mattner, sobre pruebas funcionales. Hemos realizado una publicación conjunta donde presentamos un algoritmo genético para la construcción de conjuntos de casos de prueba para el testeo de software a partir del modelo funcional del programa

Figura 6: Escenario de sincronización de semáforos en la ciudad de Málaga.

En nuestro caso estamos ahora realizando propuestas en el ámbito nacional con empresas como *Indra* en temas software, *Bordallo y Carrasco* (Arquitectos) referente a optimización de estructuras, y la empresa *Telvent* (*Schneider Electric*), para analizar las oportunidades de negocio de nuestros desarrollos. En el ámbito internacional hemos estado en contacto con empresas de Argentina (energía eólica), Uruguay (planificación), Francia (comunicación) y Alemania (pruebas), para transferir conocimientos a sus dominios de investigación.

De especial interés para este artículo es la colaboración con la empresa alemana Berner&Mattner, sobre pruebas funcionales. Hemos realizado una publicación conjunta [8] (nominada en 2012 a *best paper* en la conferencia *Genetic and Evolutionary Computation Conference*), donde presentamos un algoritmo genético para la construcción de conjuntos de casos de prueba para el testeo de software a partir del modelo funcional del programa.

En esta propuesta se generan datos de prueba para diferentes modelos extraídos de la literatura y, además, se estudia el impacto de diferentes distribuciones de probabilidad para ponderar las características del software. El algoritmo propuesto puede ser integrado en el software propietario CTE XL, de la empresa alemana Berner&Mattner, que podemos observar en la **figura 5**.

4. Líneas de trabajo futuro

Un ejercicio básico de reciprocidad obliga a la universidad a trabajar para la sociedad en general, generando plusvalías para que los ciudadanos se beneficien de nuestra investigación.

En el caso específico de las pruebas de software, la generación automática de las mismas supone un ahorro importante a las empresas de desarrollo software. El proceso de pruebas se puede descomponer en la generación de los datos de pruebas como entrada y la comprobación de la salida (problema del oráculo). Para esto último necesitamos un ingeniero de pruebas que, conociendo la funcionalidad, establezca la salida que debe producir el código que se está testando. Como trabajo futuro, se está investigando la posibilidad de generar esta salida de forma automática, como se hace por ejemplo en el proyecto RECOST. <http://recost.group.shef.ac.uk/>.

En nuestro caso, el grupo de la UMA NEO <http://neo.lcc.uma.es> ha participado en proyectos europeos como CARLINK y COADVISE, así como nacionales en relación a las pruebas de software como M*, y actualmente está llevando los resultados pasados al dominio de la movilidad en ciudad inteligente en el proyecto roadME (TIN2011-28194, 12-14).

Actualmente, se están desarrollando una serie de servicios relacionados con la vida en la ciudad, basados en las tecnologías de información y comunicación. Es lo que llamamos *Smart City* o Ciudad Inteligente.

En este sentido, el uso de técnicas bio-inspiradas para la gestión del flujo del tráfico mediante la sincronización de semáforos podría constituir uno de los aspectos más innovadores en los entornos urbanos en el futuro (ver **figura 6**). No obstante, la programación automática de semáforos requiere además de un proceso de validación de las soluciones generadas, dado que afectan a la seguridad de miles de usuarios.

Por esto, una línea de trabajo futuro que ya estamos abordando consiste en proponer una estrategia de validación basada en los Modelos de Características (*Feature Models*) para generar automáticamente diversos escenarios que permitan comprobar la robustez de los programas de semáforos. El resultado es información validada sobre el programa de semáforos que mejor se comporta para la mayoría de los escenarios posibles (días con lluvia, accidentes en autovías, piso helado, etc.).

Referencias

[1] S. Afshan, P. McMinn, M. Stevenson. Evolving Readable String Test Inputs Using a Natural Language Model to Reduce Human Oracle Cost. *International Conference on Software Testing, Verification and Validation* (ICST), 2013.

[2] E. Alba, J. F. Chicano. Management of Software Projects with GAs. *The 6th Metaheuristics International Conference* (MIC), pp. 13-18, Vienna, Austria, 2005.

[3] J. Ferrer, F. Chicano, E. Alba. Evolutionary Algorithms for the Multi-objective Test Data Generation Problem. *Software: Practice & Experience 42:* pp.1331-1362, 2012.

[4] J. Ferrer, F. Chicano, E. Alba. Benchmark Generator for Software Testers. *Artificial Intelligence Applications and Innovations 364*: pp. 378-388, Springer, 2011.

[5] J. Ferrer, F. Chicano, E. Alba. Correlation Between Static Measures and Code Coverage in Evolutionary Test Data Generation. *International Journal of Software Engineering and its Applications 4(4)*: pp. 57-79, 2010.

[6] J. Ferrer, F. Chicano, E. Alba. Dealing with Inheritance in OO Evolutionary Testing. *Proceedings of the Annual Conference on Genetic and Evolutionary Computation* (GECCO), pp. 1665-1672, 2009.

[7] J. Ferrer, F. Chicano, E. Alba. Elementary Landscape Decomposition of the Test Suite Minimization Problem. *Proceedings of the Third International Symposium on Search Based Software Engineering* (SSBSE), Springer, pp. 48-63, 2011.

[8] J. Ferrer, P.M. Kruse, F. Chicano, E. Alba. Evolutionary Algorithm for Prioritized Pairwise Test Data Generation. *Proceedings of the Annual Conference on Genetic and Evolutionary Computation* (GECCO), pp. 1213-1220, 2012.

[9] M. Gendreau, J. Potvin. *Handbook of Metaheuristics.* ISBN 978-1-4419-1665-5, 2010.

[10] M. Harman, B. F. Jones. Search-based Software Engineering. *Journal of Information and Software Technology, 43*(14): pp. 833-839, 2001.

[11] G. Myers, T. Badgett, C. Sandler. *The Art of Software Testing.* John Wiley and Sons, 2011. ISBN: 1118031962.

[12] Y. Zhang, M. Harman, A. Mansouri. The Multi-objective Next Release Problem. *Proceedings of the International Conference on Genetic and Evolutionary Computation Conference* (GECCO), pp. 1129-1137, 2007.

Notas

[1] Los problemas NP-difíciles son aquellos al menos tan difíciles como los NP. Actualmente no se conoce ningún algoritmo que resuelva tales problemas en tiempo polinómico y ahí reside su dificultad.

[2] <http://crestweb.cs.ucl.ac.uk/resources/sbse_repository/>.

[3] Una *test suite* es un conjunto de casos de prueba cuya función es determinar si el requisito de una aplicación se ha satisfecho parcial o totalmente.

[4] Un resolutor de satisfacibilidad o *SAT-solver* es un algoritmo capaz de resolver eficientemente problemas de satisfacibilidad de clausulas (SAT). El problema SAT fue el primero en ser identificado como perteneciente a la clase de complejidad NP-completo.

Priorización de casos de prueba: Avances y retos

Ana Belén Sánchez Jerez, Sergio Segura Rueda, Antonio Ruiz-Cortés
Departamento de Lenguajes y Sistemas Informáticos, Universidad de Sevilla

<{anabsanchez,sergiosegura,aruiz}@us.es>

***Resumen**: La priorización de pruebas consiste en establecer un orden de ejecución para los casos de prueba que permita alcanzar un determinado objetivo. Por ejemplo, es posible reordenar los casos de prueba para detectar fallos lo antes posible o conseguir un determinado nivel de cobertura de código cuanto antes. En este artículo, presentamos y clasificamos de forma novedosa las propuestas de priorización de pruebas presentadas hasta la fecha. Además, destacamos algunos de los retos de la investigación en este área.*

***Palabras clave**: Detección de fallos, estrategias de priorización, pruebas de software, priorización de casos de prueba.*

Autores

Ana Belén Sánchez Jerez es Ingeniera en Informática y Máster en Ingeniería y Tecnología del Software por la Universidad de Sevilla. Actualmente, trabaja como Personal Investigador en Formación en el grupo Ingeniería del Software Aplicada de la Universidad de Sevilla donde se encuentra iniciando su tesis doctoral en el campo de la automatización de pruebas en sistemas de alta variabilidad.

Sergio Segura Rueda es Profesor Contratado Doctor en la Universidad de Sevilla donde trabaja como profesor a tiempo completo desde 2006. Es miembro del grupo de investigación Ingeniería del Software Aplicada donde investiga, entre otros temas, en la automatización de pruebas. También colabora como revisor habitual en diversas conferencias y revistas de Ingeniería del Software.

Antonio Ruiz-Cortés es Profesor Titular y director del grupo Ingeniería del Software Aplicada (ISA, <http://www.isa.us.es/>) de la Universidad de Sevilla. Obtuvo su doctorado en Ingeniería en Informática por la Universidad de Sevilla. Sus actuales líneas de investigación incluyen computación orientada a servicios, líneas de productos software y gestión de procesos de negocio.

1. Introducción

Ejecutar todos los casos de prueba de una aplicación compleja puede llegar a requerir horas, días o incluso semanas [4][18][19].

Esto puede deberse a diversas causas. Por ejemplo, en los ecosistemas modernos el número de pruebas se cuenta por millares, lo que incrementa el tiempo de ejecución. Como referencia, Eclipse tiene más de 40.000 casos de prueba [20].

Por otra parte, la ejecución de las pruebas no es siempre automática (por ej. interfaces de usuario) y en ocasiones son computacionalmente muy costosas [21].

Estos aspectos son especialmente críticos durante las pruebas de regresión cuando las pruebas deben ejecutarse repetidamente cada vez que el software sufre algún cambio relevante. En este escenario, las restricciones de presupuesto y tiempo pueden impedir la ejecución completa de una *suite* de pruebas [4][17]. Además, reducir el número de pruebas no es siempre una opción pues aumentan las probabilidades de que el programa contenga defectos.

Desde que Rothermel et al. introdujesen el concepto de priorización en 1999 [22], muchos autores han estudiado la priorización de casos de prueba como una de las alternativas para aumentar la efectividad de las pruebas. La idea es simple: reordenar los casos de prueba de manera que se ejecuten primero aquellos que permitan maximizar un determinado objetivo de rendimiento.

Esta idea es aplicable a todo tipo de pruebas tanto unitarias como de integración y sistema. Por ejemplo, podríamos acelerar la detección de fallos probando primero los componentes más complejos. Si se detectaran fallos, éstos podrían ser corregidos cuanto antes reduciendo el tiempo total necesario para pruebas y depuración. Además, si la ejecución de las pruebas se detuviese por cualquier razón, intencionada o no, sabríamos que los componentes más críticos ya han sido probados.

En este artículo, se introducen y clasifican las principales propuestas de priorización y los retos identificados en el área.

2. Clasificación de propuestas de priorización de pruebas

Proponemos clasificar las contribuciones en el área de priorización de pruebas en los siguientes términos:

- **Objetivo.** Este es el fin último de la propuesta de priorización. Por ejemplo, el objetivo más habitual es acelerar la detección de fallos.
- **Estrategia.** Criterio de ordenación utilizado para alcanzar un objetivo de priorización. Dado un objetivo, se pueden definir distintas estrategias para alcanzarlo. Por ejemplo, una posible estrategia para acelerar la detección de fallos es probar primero aquellos componentes más complejos. Otra posible estrategia es reordenar los casos de prueba de manera que se ejecuten primero aquellas pruebas que validan los componentes que han demostrado ser más propensos a fallos en el pasado.
- **Propuesta.** Es el método concreto usado para implementar la estrategia de priorización. Dada una estrategia de priorización se pueden encontrar distintas propuestas para implementarla. Por ejemplo, para reordenar los casos de prueba en función de la complejidad del software podríamos tomar como referencia el conocimiento del desarrollador o el uso de métricas que midan su complejidad (por ej. complejidad ciclomática).

La **tabla 1** muestra un resumen de los objetivos, estrategias y algunas de las propuestas de priorización identificados en la literatura. Note que algunas estrategias son usadas con distintos objetivos.

3. Objetivos de priorización

Identificamos cuatro grandes objetivos de priorización en la literatura:

- **Objetivo 1. Acelerar la detección de fallos.** Este es uno de los objetivos de priorización más estudiados [1][6][17]. Para su aplicación se ordenan los casos de prueba de acuerdo a su habilidad para detectar fallos. De este modo, las primeras pruebas en ejecutarse serán aquellas con mayor probabilidad de detectar comportamientos inesperados en el software.
- **Objetivo 2. Acelerar la detección de fallos críticos.** Este objetivo considera la gravedad de los fallos como un factor a tener en cuenta. Por ejemplo, los sistemas de registro de fallos como BugZilla[1] identifican 7 niveles de relevancia para un fallo: bloqueador, crítico, mayor, normal, menor, trivial y me-

> “La idea es simple: reordenar los casos de prueba de manera que se ejecuten primero aquellos que permitan maximizar un determinado objetivo de rendimiento”

Objetivos	Estrategias	Propuestas
1. Acelerar la detección de fallos	Basada en responsable de pruebas	[10]
	Basada en histórico de fallos	[7], [8], [11]
	Basada en histórico de cambios	[25]
	Basada en requisitos del cliente	[10]
	Basada en complejidad del software	[10]
	Basada en la similitud de las pruebas	[5], [11],[19]
	Basada en acoplamiento de componentes	[9]
2. Acelerar la detección de fallos críticos	Basada en histórico de fallos	[24]
	Basada en requisitos de cliente	[3]
	Basada en complejidad del software	[3]
3. Acelerar la cobertura del código	Basada en la similitud de las pruebas	[5], [11], [19]
	Basada en la cobertura del código	[4], [5], [8]
4. Minimizar los costes asociados a las pruebas	Basada en costes	[6], [7], [24]

Tabla 1. Clasificación de objetivos, estrategias y propuestas de priorización.

jora. El nivel de gravedad de un fallo vendrá determinado por la aplicación que se está probando. Por ejemplo, en ciertas aplicaciones los fallos más críticos serán aquellos que afecten a aspectos de seguridad (por ej. banca) mientras que en otras pueden ser aquellos que afecten a su funcionalidad básica (por ej. controlador hardware). Así, para alcanzar este objetivo los casos de prueba se reordenan en función de su habilidad esperada para detectar fallos críticos [16][17].

■ **Objetivo 3. Acelerar la cobertura del código.** La finalidad de este objetivo es ejecutar los casos de prueba en un orden que permita alcanzar el nivel deseado de cobertura de código cuanto antes [1][16][17]. Este objetivo está muy relacionado con el objetivo 1, ya que se basa en la suposición de que maximizando la cobertura del código se incrementará la probabilidad de maximizar la detección de fallos. Para su consecución, los casos de pruebas se ordenan de acuerdo al porcentaje de código que cubren, ejecutando primero aquellos que permiten alcanzar una mayor cobertura de código.

■ **Objetivo 4. Minimizar los costes asociados a las pruebas.** No todas las pruebas requieren el mismo coste de configuración, ejecución y validación [1][7][16][17]. El coste de una prueba, por ejemplo, puede medirse en términos del tiempo necesario para ejecutarla. Otra medida que puede considerarse es el coste económico de la configuración, ejecución y validación de la prueba; esto puede reflejar el coste del hardware, salarios, coste de materiales requeridos, etc. Así, se ejecutarán antes las pruebas que conlleven menor coste. Este objetivo suele perseguirse conjuntamente con otros objetivos de priorización. Por ejemplo, es muy usual ver asociado este objetivo al objetivo 1, con el fin de conseguir minimizar los costes asociados a las pruebas sin perder la capacidad de detección de fallos. También, se puede ver asociado al objetivo 3, minimizando costes a la vez que se mantiene una buena cobertura de código.

4. Estrategias y propuestas de priorización

En esta sección, presentamos algunas de las estrategias y propuestas de priorización más citadas en la literatura. Para facilitar su comprensión haremos referencia al ejemplo mostrado en la **tabla 2**, la cual muestra información real sobre cuatro de los módulos de la solución de comercio electrónico de código abierto Prestashop 1.5 [23]. Para cada módulo, se muestran el número de fallos[2] encontrados y el número de cambios[3] realizados desde el 01/05/2013 al 30/06/2013.

4.1. Estrategia basada en el responsable de pruebas

Esta estrategia se utiliza para acelerar la detección de fallos (objetivo 1). El responsa-

	Paypal	Cesta compra (blockcart)	Categorías productos (productscategory)	Productos favoritos (favoriteproducts)
Número fallos	27	37	15	3
Número cambios	13	16	1	3

Tabla 2. Número de fallos y cambios en algunos de los módulos de Prestashop 1.5.

ble de pruebas, según su experiencia, asigna prioridades de ejecución a los casos de prueba de acuerdo a su probabilidad para detectar fallos [1][17].

Por ejemplo, supongamos que el responsable de pruebas asigna las siguientes prioridades a las pruebas de los módulos de la **tabla 2**: *paypal*=5, *cesta compra*=9, *categorías productos*=8 y *productos favoritos*=2. En este caso, se ejecutarían primero las pruebas del módulo *cesta compra* seguidas de las pruebas de *categorías, paypal* y *productos favoritos*.

Es importante destacar que esta estrategia no es sólo aplicable a las pruebas unitarias sino también, por ejemplo, a las de integración. Así, la prueba de integración *{cesta compra + categorías productos}* (9+8=17) podría tener mayor prioridad que la prueba de integración *{paypal+cesta compra+productos favoritos}* (5+9+2=16).

Siguiendo esta estrategia, en [10] se propone ordenar las pruebas basándose en varios factores, entre otros, completitud, trazabilidad e impacto de fallos. Para ello, el responsable de pruebas asigna pesos numéricos ([1-10]) que determinan el orden de ejecución de las pruebas.

4.2. Estrategia basada en histórico de fallos

Esta estrategia es utilizada para acelerar la detección de fallos en general (objetivo 1) o fallos críticos en particular (objetivo 2). Las pruebas se ordenan en base a un histórico de fallos de versiones previas, ejecutándose antes aquellas pruebas que validan los componentes que han sido más propensos a fallos en versiones anteriores [1][16][17].

Por ejemplo, supongamos que vamos a probar una nueva versión de Prestashop siguiendo esta estrategia. De acuerdo a los datos de la **tabla 2**, ejecutaríamos en primer lugar los casos de prueba de *cesta compra* pues es el módulo que ha sido más propenso a fallos en versiones anteriores (37 fallos), seguido del módulo *paypal* (27 fallos), *categorías productos* (15) y *productos favoritos* (3).

En [8] se asignan prioridades binarias a los casos de prueba en función de si han revelado fallos (1) o no (0) en versiones anteriores del software. En [7], se reordenan las pruebas de sistemas configurables considerando su capacidad de detección de fallos (utilizando históricos de versiones anteriores) y también el coste de configuración y preparación de las pruebas. Simons et al. [11] agrupan y priorizan las pruebas en función de los datos reflejados en el histórico de fallos.

En [24] los autores proponen acelerar la detección de fallos críticos a partir de información sobre el coste de las pruebas así como el número y la gravedad de los fallos detectados en las últimas pruebas de regresión realizadas. Después, los autores aplican un algoritmo genético para encontrar el orden con el mayor ratio de fallos críticos detectados por el coste de las pruebas.

4.3. Estrategia basada en histórico de cambios

Esta estrategia tiene como objetivo acelerar la detección de fallos (objetivo 1). Las pruebas se ejecutan de acuerdo al número de cambios realizados en sus componentes en versiones previas.

La estrategia se basa en el hecho de que los cambios realizados a componentes de un sistema, a menudo, pueden añadir un nuevo fallo en el código [1][16][17]. Por ejemplo, en función de los cambios realizados en los módulos de la **tabla 2**, ejecutaríamos antes las pruebas del módulo *cesta compra* (16 cambios), seguidas de las pruebas de los módulos *paypal* (13), *productos favoritos* (3) y *categorías* productos (1).

Sherriff et al. [25] proponen una metodología para priorizar los casos de prueba de regresión mediante la recopilación de los registros de cambios del software. Además, esta propuesta identifica grupos de ficheros que históricamente tienden a cambiar juntos.

4.4. Estrategia basada en requisitos del cliente

Esta estrategia es utilizada para acelerar la detección de fallos (objetivos 1 y 2). Utiliza los requisitos del cliente para asignar prioridades a los componentes del sistema [1][16][17].

Por ejemplo, supongamos que el cliente de una tienda de comercio electrónico considera como partes relevantes del sistema (indicadas en los requisitos) aquéllas relacionadas con el módulo de *cesta compra*, a la que asigna prioridad=10, seguido del módulo *categorías productos* (prioridad=8), *productos favoritos* (prioridad=4) y *paypal* (prioridad=3). Estas prioridades establecerán el orden de ejecución de las pruebas.

En [3] y [10] los autores proponen acelerar la detección de fallos priorizando los casos de prueba de acuerdo a la prioridad de los requisitos asignada por el cliente. A mayor valor de los factores, mayor será la prioridad de ejecución de la prueba relacionada con ese requisito.

4.5. Estrategia basada en complejidad del software

Esta estrategia persigue acelerar la detección de fallos (objetivos 1 y 2). Reordena la ejecución de las pruebas basándose en la complejidad de implementación del software. Este dato será proporcionado por los desarrolladores del programa de acuerdo a su experiencia durante la fase de implementación. También podrían emplearse métrcas de código (por ej. complejidad ciclomática) para obtener medidas cuantitativas de la complejidad.

Esta estrategia se basa en la hipótesis de que aquellos componentes más difíciles de desarrollar serán más propensos a fallos. Así, se ejecutan antes los casos de prueba de los componentes con mayor complejidad [1][16][17].

En [3] y [10] se propone priorizar los casos de prueba asignando pesos ([1-10]) que midan la complejidad del software (o el requisito) que validan. A mayor peso, mayor será la prioridad de ejecución del caso de prueba asociado.

4.6. Estrategia basada en la similitud de las pruebas

El objetivo de esta estrategia puede ser tanto acelerar la detección de los fallos (objetivo 1) como acelerar la cobertura del código (objetivo 3).

Existen estudios que demuestran que los casos de prueba similares (aquellos que ejecutan porciones de código comunes) son redundantes desde el punto de vista del descubrimiento de nuevos fallos. Sin embargo, las pruebas que difieren más entre ellas tienen más probabilidades de detectar comportamientos inesperados [1][16][17][19]. Así, en esta estrategia se ejecutan primero las pruebas que difieren más entre sí.

En [5] se presenta una propuesta de priorización basada en la similitud de las pruebas en función del código que cubren. En [11] se propone priorizar las pruebas de acuerdo a sus similitudes agrupándolas según su habilidad para detectar fallos en versiones anteriores. Henard et al. [19] también proponen la priorización basada en similitudes (se agrupan las pruebas que validan los mismos componentes) para generar casos de prueba de líneas de productos software.

4.7. Estrategia basada en acoplamiento entre componentes

Esta estrategia se utiliza para acelerar la detección de fallos (objetivo 1). Se basa en las dependencias de los componentes del sistema para priorizar los casos de prueba.

Este tipo de estrategia se suele ver reflejada en propuestas basadas en modelos (modelos UML, grafos, modelos de estados, etc.), que aprovechan su estructura para representar los componentes del sistema y las dependencias entre ellos [1][9]. Así, se da priori-

> "La estrategia basada en la cobertura del código software persigue ejecutar la mayor cantidad de código posible cuanto antes (objetivo 3). Para ello, se reordenan las pruebas según la cobertura de código alcanzada en ejecuciones previas, ejecutándose primero las pruebas con mayor cobertura"

dad de ejecución a las pruebas que contienen mayor acoplamiento entre sus componentes, ya que ésto puede representar un indicio de aparición de fallos.

Tahat et al. [9] presentan un trabajo de priorización que utiliza modelos que describen el comportamiento del sistema a través de un conjunto de estados y transiciones entre estados. Para la reordenación, los autores asignan una prioridad alta a las pruebas que ejecutan las transiciones modificadas (añadidas o eliminadas) en el modelo y una prioridad baja a las pruebas que no ejecutan ninguna transición modificada.

4.8. Estrategia basada en la cobertura del código software

Esta estrategia persigue ejecutar la mayor cantidad de código posible cuanto antes (objetivo 3). Para ello, se reordenan las pruebas según la cobertura de código alcanzada en ejecuciones previas, ejecutándose primero las pruebas con mayor cobertura.

Por ejemplo, asumiendo porcentajes de cobertura ficticios para los módulos de la **tabla 2**, podríamos definir el siguiente orden de ejecución: casos de prueba de *paypal* (45% de cobertura), *cesta compra* (25%), *categorías productos* (20%) y *productos favoritos* (10%). Es frecuente que pruebas distintas ejerciten partes comunes del código. En estos casos, es posible asignar las prioridades teniendo en cuenta no sólo el porcentaje total de código cubierto sino también el porcentaje de código nuevo no cubierto por pruebas anteriores.

En [5] se presenta una propuesta de priorización de pruebas basada en cobertura adicional. Para ello, primero se selecciona el caso de prueba que cubra el mayor porcentaje de código, después, se selecciona el caso de prueba que cubra el mayor porcentaje de código no cubierto por el primero, y así sucesivamente.

Rothermel et al. [4] realizan varias propuestas de priorización agrupadas en las siguientes categorías: a) propuestas que ordenan las pruebas basadas en la cobertura total del código, b) propuestas que ordenan las pruebas basadas en la cobertura del código de los componentes que no han sido cubiertos previamente, y c) propuestas que ordenan las pruebas basadas en la estimación de la habilidad para revelar fallos en el código que cubren.

Por otro lado, en [8] se presenta una propuesta de priorización que otorga mayor prioridad a aquellos casos de prueba que cubran funciones que hayan sido raramente cubiertas en sesiones de pruebas anteriores.

4.9. Estrategia basada en costes

Esta estrategia se utiliza para minimizar los costes asociados a las pruebas (objetivo 4).

Cuando el coste es un factor relevante, este tipo de estrategias y las propuestas que las implementan presentan una solución efectiva. El coste de una prueba está relacionado con los recursos requeridos para configurarla, ejecutarla y validarla: tiempo de máquina, esfuerzo humano, coste de hardware, materiales, etc. Así, se ejecutan primero las pruebas que conllevan menor coste asociado.

En [6] se presenta una propuesta de priorización basada en el coste de las pruebas. El coste de una prueba lo relacionan con la suma del tiempo requerido para ejecutar la prueba y validar su salida mediante comparación con la salida esperada. Srikanth et al. [7] priorizan las pruebas de sistemas configurables basándose no solo en su capacidad de detección de fallos sino también en su coste de configuración y preparación. En [24] se propone reordenar las pruebas en base al coste de éstas en términos de tiempo de ejecución obtenido de las últimas pruebas de regresión realizadas.

4.10. Estrategia multiobjetivo

Existen propuestas de priorización de pruebas denominadas multiobjetivo o multifaceta que utilizan varias estrategias a la vez para conseguir ordenar las pruebas de modo que se logren uno o varios objetivos de rendimiento.

Por ejemplo, Srikanth et al. [3] proponen una estrategia multiobjetivo que explora tres factores: 1) prioridad asignada por el cliente, 2) complejidad de los requisitos y 3) volatilidad de los requisitos. A su vez, esta propuesta persigue dos objetivos: identificar fallos críticos lo antes posible y minimizar el coste de la priorización de las pruebas.

Otro ejemplo se presenta en [10] donde se consideran 6 factores de priorización: prioridad de los requisitos asignada por el cliente, complejidad de la implementación del código percibida por el desarrollador, cambios en los requisitos, impacto de fallos, completitud y trazabilidad.

5. Métricas de evaluación

Se han propuesto diversas métricas para evaluar la efectividad de las propuestas de priorización. La métrica más utilizada [1] es la conocida por sus siglas **APFD** (*Average Percentage of Faults Detected*).

La métrica APFD [12] evalúa la rapidez con la que son detectados los fallos durante el proceso de pruebas. APFD calcula la media ponderada del porcentaje de fallos detectados sobre el conjunto de pruebas. Para definir formalmente APFD, consideraremos **T** un conjunto de pruebas que contiene **n** casos de prueba, y **F** un conjunto de fallos detectados por **T**. **TFi** representa la posición del primer caso de prueba del conjunto ordenado **T'** que detecta el fallo **i**. Siguiendo esta definición, la métrica APFD para el conjunto T' vendría dada por la ecuación:

$$APFD = 1 - \frac{TF_1 + TF_2 + \_ + TF_n}{n \times m} + \frac{1}{2n}$$

APFD devuelve un valor entre 0 y 1, donde los valores más altos indican una velocidad mayor de detección de fallos del conjunto de pruebas T'.

Existen otras métricas para evaluar la priorización como son: *ASFD (Average Severity of Faults Detected)* [13], *TPFD (Total Percentage of Faults Detected)* [10], *APFD(c) (Average Percentage of Faults Detected per Cost)* [14], *NAPFD (Normalized APFD)* [15] y *CE (Coverage Effectiveness)* [12].

6. Estadísticas y tendencias

A continuación, se muestran algunas estadísticas y tendencias de las propuestas de priorización de pruebas tomadas de un reciente estudio [1]. La **figura 1** muestra información sobre el número de trabajos de priorización (de revistas y de congresos) publicados hasta 2010. Lo más destacable es la tendencia ascendente de estos trabajos iniciada en 2004 (4 publicaciones) y que

> “Son necesarias nuevas propuestas de priorización compatibles con restricciones de variabilidad que permitan priorizar las pruebas de familias de productos”

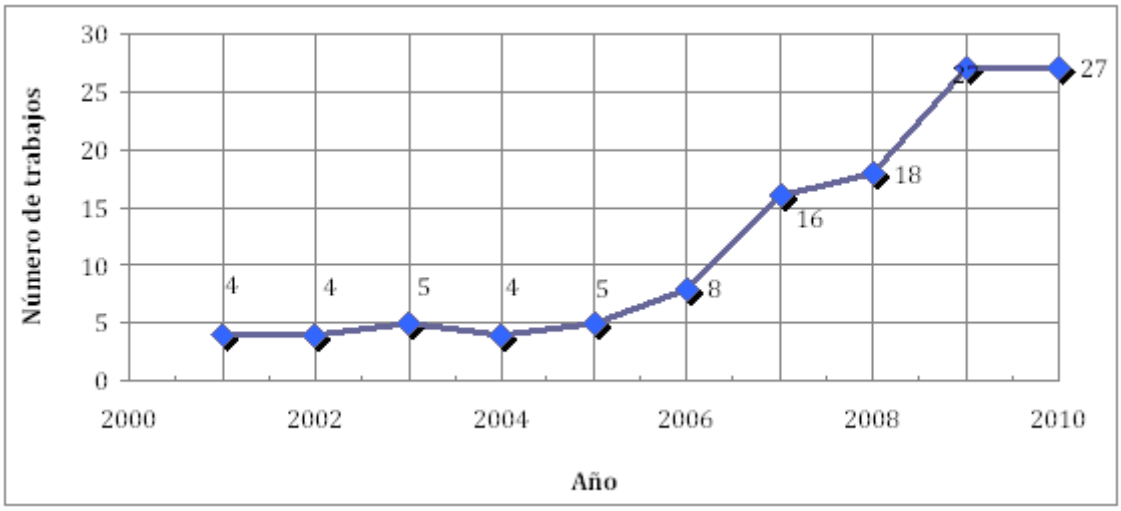

Figura 1. Número de trabajos de priorización por año.

llega hasta 2010 (27 publicaciones). El interés sobre priorización de pruebas parece estar en auge.

La **figura 2** ilustra la distribución de las propuestas de priorización de pruebas más investigadas [1].

Existe una gran variedad de propuestas, siendo los métodos basados en cobertura los más utilizados. Además, las propuestas basadas en modelos están aumentando en los últimos años.

7. Retos

A continuación, destacamos algunos de los retos en el área de priorización de pruebas identificados en estudios recientes [1][16] así como derivados de nuestro propio trabajo investigador:

- **Estudios comparativos.** Para seleccionar la propuesta de priorización que mejor se adapte a cada problema son necesarios estudios comparativos que permitan conocer la aplicabilidad y efectividad de cada propuesta.
- **Priorización de pruebas en sistemas de alta variabilidad.** Las pruebas en sistemas de alta variabilidad, como las líneas de productos software, son un gran reto debido al elevado número de variantes (potencialmente millones) que deben probarse. En este tipo de sistemas la priorización de pruebas po-

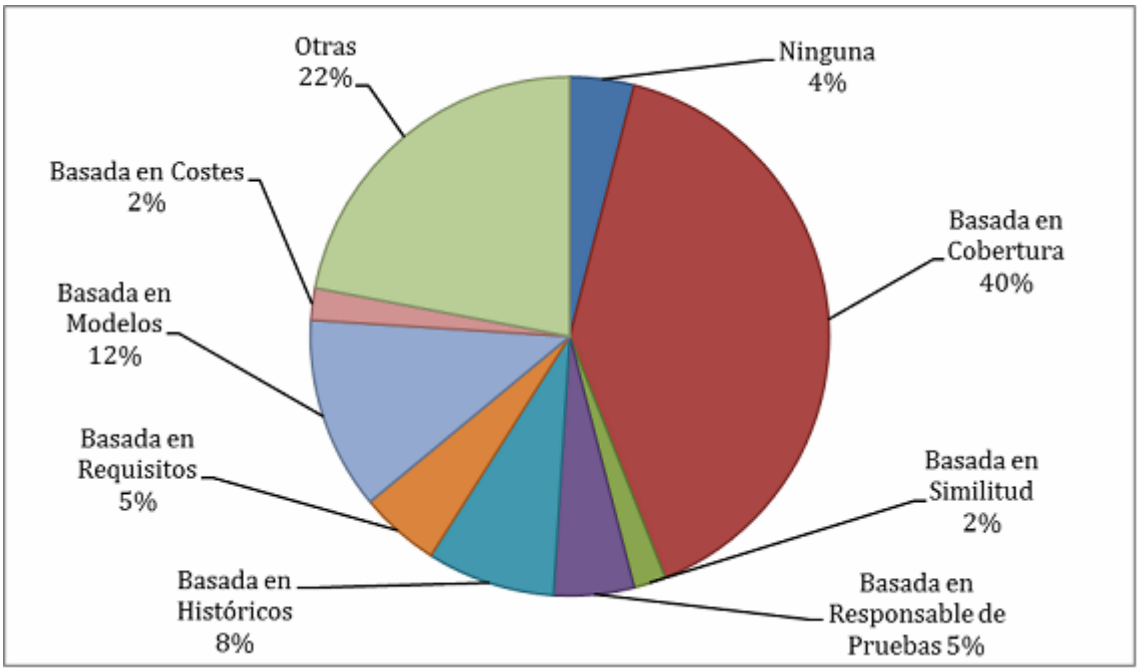

Figura 2. Distribución de propuestas de priorización.

dría implicar beneficios significativos. Son necesarias nuevas propuestas de priorización compatibles con restricciones de variabilidad que permitan priorizar las pruebas de familias de productos.

- **Métricas de evaluación.** Una cuarta parte de los trabajos de priorización no utilizan ninguna métrica para evaluar la efectividad de sus propuestas. No es posible juzgar la eficiencia de las propuestas sin el uso de una métrica que las evalúe y sin la comparación con otros métodos de priorización. Así, es necesario proponer nuevas métricas o promover el uso de las existentes.
- **Herramientas de priorización.** Existe una gran carencia de herramientas de priorización que permitan automatizar la reordenación dinámica de los casos de prueba. Sería especialmente interesante que este tipo de herramientas se pudieran integrar en entornos de desarrollo como Eclipse.
- **Priorización de múltiples conjuntos de prueba.** Las propuestas de priorización asumen explícitamente que hay solo un conjunto de pruebas. Sin embargo, es posible encontrar aplicaciones, especialmente aquellas desarrolladas de forma distribuida, con más de un conjunto de pruebas. Hasta la fecha, sólo hemos encontrado una propuesta de priorización que resuelva el problema de múltiples conjuntos [16].
- **Casos de estudio.** El 64% de los trabajos sobre priorización utilizan datos de proyectos industriales, que reflejan problemas reales de la industria [1]. El resto de trabajos utilizan proyectos de laboratorio que conducen a resultados significativos si no son demasiado pequeños. Es por ello que destacamos la necesidad de utilizar proyectos grandes de laboratorio o industriales para realizar el estudio de la priorización de pruebas.

8. Conclusiones

El orden en el que se ejecutan las pruebas es más relevante a medida que aumenta la complejidad y el tamaño de las aplicaciones software. Elegir el orden adecuado nos permite alcanzar nuestros objetivos antes, disminuyendo el tiempo total invertido en pruebas y depuración.

En este artículo, presentamos y clasificamos los objetivos, estrategias y principales propuestas de priorización. Además, mostramos algunos de los retos identificados en la literatura y que vertebrarán la futura investigación en el área.

Referencias

[1] C. Catal, D. Mishra. "Test case prioritization: a systematic mapping study". *Software Quality Journal*, DOI: 10.1007/s11219-012-9181-z, 2012.

[2] S. Elbaum, G. Rothermel, S. Kanduri, A. G. Malishevsky. "Selecting a cost-effective test case prioritization technique". *Software Quality Control Journal*, *12, 3*, pp. 185-210, 2004.

[3] Hema Srikanth, Laurie Williams. "*Requirements-Based Test Case Prioritization*". Department of Computer Science, North Carolina State University, 2005.

[4] Gregg Rothermel, Roland H. Untch, Chengyun Chu, Mary Jean Harrold. "Prioritizing Test Cases for Regression Testing". *IEEE Transactions on Software Engineering, 27, 10*, pp. 929-948, 2001.

[5] David Leon, Andy Podgurski. "A Comparison of Coverage-Based and Distribution-Based Techniques for Filtering and Prioritizing Test Cases". *Symposium on Software Reliability Engineering*, pp. 442-453, 2003.

[6] Alexey G. Malishevsky, Joseph R. Ruthruff, Gregg Rothermel Sebastian Elbaum. "*Cost-cognizant Test Case Prioritization*". Technical Report, Department of Computer Science and Engineering, University of Nebraska-Lincoln, 2006.

[7] Hema Srikanth, Myra B. Cohen, Xiao Qu. "Reducing Field Failures in System Configurable Software: Cost-Based Prioritization". *Conference On Software Reliability Engineering,* 2009.

[8] Jung-Min Kim, Adam Porter. "A history-based test prioritization technique for regression testing in resource constrained environments". *Conference on Software Engineering,* 2002.

[9] Luay H. Tahat, Bogdan Korel, Mark Harman, Hasan Ural. "Regression test suite prioritization using system models". *Software, Verification & Reliability Journal, 22, 7*, pp. 481-506, 2012.

[10] R. Krishnamoorthi, S. A. Sahaaya Arul Mary. "Factor Oriented Requirement Coverage based System Test Case Prioritization of New and Regression Test Cases". *Information and Software Technology, 51, 4*, pp. 799-808, 2009.

[11] Cristian Simons, Emerson Cabrera Paraiso. "Regression Test Cases Prioritization using Failure Pursuit Sampling". *Proceedings of the ISDA*, pp. 923-928, 2010.

[12] Gregory M. Kapfhammer, Mary Lou Soffa. "Using Coverage Effective- ness to Evaluate Test Suite Prioritizations". *Workshop Empirical Assessment of Software Engineering Languages and Technologies*, pp. 19-20, 2007.

[13] H. Srikanth, L. Williams. "On the economics of requirements-based test case prioritization". *International Workshop on Economics-driven Software Engineering Research*, pp. 1-3, 2005.

[14] Zengkai Ma and Jianjun Zhao. "Test Case Prioritization Based on Analysis of Program Structure". Asia-Pacific Software Engineering Conference, pp. 471-478, 2008.

[15] X. Qu, M. B. Cohen, K. M. Woolf. "Combinatorial Interaction Regression Testing: A Study of Test Case Generation and Prioritization". *Conference on Software Maintenance*, pp. 255-264, 2007.

[16] Siripong Roongruangsuwan, Jirapun Daengdej. "Test Case Prioritization Techniques". *Theoretical and Applied Information Technology Journal, 18, 2*, 2010.

[17] S. Yoo, M. Harman. "Regression Testing Minimisation, Selection and Prioritisation: A Survey". *Software Testing, Verification and Reliability Journal, 22, 2*, pp. 67-120, 2007.

[18] S. Elbaum, P. Kallakuri, A.G. Malishevsky, G. Rothermel, S. Kanduri. "Understanding the effects of changes on the cost-effectiveness of regression testing techniques". *Software Testing, Verification, and Reliability Journal, 13, 2*, pp. 65-83, 2003.

[19] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, Yves Le Traon. "*Bypassing the Combinatorial Explosion: Using Similarity to Generate and Prioritize T-wise Test Suites for Large Software Product Lines*", Technical report, 2012.

[20] Martin Fagereng Johansen, Øystein Haugen, Franck Fleurey, Erik Carlson, Jan Endresen, and Tormod Wien. "A Technique for Agile and Automatic Interaction Testing for Product Lines". *Conference Testing Software and Systems*, pp. 39-54, 2012.

[21] Glenford J. Myers. "*The Art of SoftwareTesting*". John Wiley & Sons, Inc, Second Edition, 2004. ISBN 0-471-46912-2.

[22] Gregg Rothermel, Roland H. Untch, Chengyun Chu, Mary Jean Harrold. "Test Case Prioritization: An Empirical Study". *Conference on Software Maintenance*, pp. 179-188, 1999.

[23] Prestashop. <http://www.prestashop.com/>. Último acceso: junio de 2013.

[24] Yu-Chi Huang, Chin-Yu Huang, Jun-Ru Chang, Tsan-Yuan Chen. «Design and Analysis of Cost-Cognizant Test Case Prioritization Using Genetic Algorithm with Test History». *Computer Software and Applications Conference*, pp. 413-418, 2010.

[25] Mark Sherriff, Mike Lake, Laurie Williams. "Prioritization of Regression Tests using Singular Value Decomposition with Empirical Change Records". *Symposium on Software Reliability Engineering*, pp. 81-90, 2007.

Notas

[1] <http://www.bugzilla.org/>.

[2] <http://forge.prestashop.com/secure/Dashboard.jspa>.

[3] <https://github.com/PrestaShop>.

Utilización de MDE para la prueba de sistemas de información web

Federico Toledo Rodríguez[1] , Macario Polo Usaola[2], Beatriz Pérez Lamancha[2]

[1] Abstracta, Montevideo (Uruguay); [2]Universidad de Castilla-La Mancha, Ciudad Real (España)

<ftoledo@abstracta.com.uy>,
<{macario.polo, beatriz.plamancha}@uclm.es>

Resumen: *Las pruebas de software son esenciales tanto en el desarrollo de nuevos sistemas como durante su mantenimiento. En el caso concreto de los sistemas de información (aquellos en los cuales el componente principal es la base de datos, cuyas entidades el usuario gestiona mediante distintas interfaces), el componente más estable es la propia base de datos, cuyo diseño tiene una influencia directa en el diseño de las diversas aplicaciones que la gestionan. Por ello, también es interesante basar el diseño de las pruebas de los sistemas de información en las estructuras de la base de datos: es decir, en sus entidades, relaciones, restricciones, etc., para poder verificar así que son manipuladas correctamente por los programas que la gestionan. En este contexto podemos considerar que hay dos formas principales para construir casos de prueba: (1) especificando un modelo del sistema y derivando pruebas automáticamente para comprobar que se verifica esa especificación, o (2) especificando directamente las pruebas, para lo que el* tester *ha de mantener el modelo del sistema en su cabeza. En este artículo describimos un enfoque dirigido por modelos para generar pruebas basadas en el modelo de datos de la base de datos, y que permite seguir ambos caminos de forma integrada, describiendo el modelo del sistema para generar pruebas, o describiendo directamente las pruebas. Luego, este modelo de pruebas se utiliza para generar una infraestructura de ejecución de pruebas, incluyendo casos de prueba y datos de prueba según la especificación proporcionada. Para facilitar la especificación del modelo del sistema, se propone además la utilización de técnicas de ingeniería inversa sobre el esquema de la base de datos.*

Palabras clave: *Datos de prueba, pruebas basadas en modelos, pruebas de sistemas de información, pruebas de software.*

Autores

Federico Toledo Rodríguez es miembro fundador de Abstracta, empresa uruguaya dedicada a brindar herramientas y servicios de *testing tercerizado*. Actualmente es estudiante de doctorado en Informática en la Universidad de Castilla-La Mancha, en el Departamento de Tecnologías y Sistemas de Información. Su investigación está relacionada con la automatización de pruebas en sistemas de información.

Macario Polo Usaola es profesor en el Departamento de Tecnologías y Sistemas de Información en la Universidad de Castilla-La Mancha. Su investigación está relacionada con la automatización de tareas en pruebas de software. Tiene un doctorado en Informática por la Universidad de Castilla-La Mancha. Además, es escritor y ha publicado varias novelas.

Beatriz Pérez Lamancha es profesora de Ingeniería de Software en la Universidad de la República, en Uruguay. Es Doctora en Informática por la Universidad de Castilla-La Mancha. Su investigación está relacionada con la automatización de las pruebas de software, especialmente en el contexto de técnicas basadas en modelos.

1. Introducción

Un *sistema de información* (SI) o *aplicación orientada a datos* es un sistema cuyo componente central es la base de datos (BD) o un conjunto de archivos permanentes [1]. La información se presenta a los usuarios mediante distintas aplicaciones compuestas por diversas capas. La estructura de estas aplicaciones guarda una correspondencia clara con el modelo de la BD: de hecho, hay diversos trabajos que, mediante alguna técnica de ingeniería inversa, obtienen el esquema conceptual de la BD y lo utilizan como modelo de la capa de dominio de nuevas aplicaciones de gestión de los datos [2][3][4].

Aunque la propia BD ya incorporará su propio conjunto de restricciones, la lógica de los programas asociados a ella debe también contemplarlas y manejarlas correctamente. En especial, se deben considerar las operaciones de creación, lectura, actualización y borrado de las entidades (conocido como patrón CRUD del inglés *create, read, update, delete*), ya que en cualquier SI la mayoría de las funcionalidades estarán formadas por este tipo de operaciones sobre las distintas entidades del modelo de datos.

Este artículo se centra en el diseño de casos de prueba para sistemas de información a partir de la BD, con los que se prueba el comportamiento de las aplicaciones que la acceden, comprobando si son capaces de manejar las particularidades de las estructuras definidas de forma adecuada.

En la última década la utilización de modelos para las diferentes actividades del desarrollo de software se hace cada vez más común. Las pruebas de software no quedan fuera de esta tendencia, destacándose el *Testing Dirigido por Modelos* (*Model-Driven Testing*, MDT) [5], que se refiere a pruebas basadas en modelos donde los casos de prueba son generados automáticamente desde artefactos de software mediante transformación de modelos, esto es, la aplicación de técnicas de *Model-Driven Engineering* (MDE) al *testing*. Nuestra propuesta aplica un enfoque MDT.

Básicamente se puede considerar que hay dos formas principales de construir casos de prueba: 1) especificando un modelo del sistema y derivando pruebas en forma automática de forma tal que verifiquen esa especificación, o 2) especificando directamente las pruebas, donde el *tester* es el que contiene el modelo del sistema en su cabeza y diseña él mismo las pruebas para verificarlo [6].

A continuación, en la **sección 2**, se presenta el esquema general propuesto para probar SI. Luego, en la **sección 3** se presentan los distintos artefactos utilizados en este esquema, así como los mecanismos de derivación de pruebas. Por último, se presentan los trabajos relacionados en la **sección 4**, y las conclusiones y trabajo futuro en la **sección 5**.

2. Marco para la generación de pruebas para SI

La **figura 1** ilustra nuestro trabajo: el *tester* puede cargar información del sistema para a partir de ahí generar pruebas en forma automática, cargar directamente pruebas en el modelo de pruebas, o ambas.

El resultado deseado en ambos casos es el mismo: generar un conjunto de casos de pruebas ejecutables sobre el sistema que sean capaces de dar un veredicto sobre su comportamiento.

Trabajar con modelos (y manejando así un nivel de abstracción más alto) presenta ciertas ventajas tales como no necesitar de programadores para las tareas de pruebas de

> "Para la generación del código de pruebas es necesario agregar información al modelo que mapee los elementos de la aplicación a los elementos de dicho modelo"

sistema, menor costo de mantenimiento, más facilidad de comprensión, etc. [7]. Encierra también algunos peligros, como la posibilidad de perder detalles que son visibles sólo a más bajo nivel.

Este trabajo aprovecha las ventajas y trata de minimizar las desventajas mediante la automatización casi completa del proceso y mediante la conservación de la correspondencia entre los elementos de todos los modelos y los niveles de abstracción.

Como apoyo a la construcción del modelo del sistema, se sugiere el uso de técnicas de ingeniería inversa.

Los elementos cuyo modelado debe considerarse, tal como vemos en la **figura 1**, son:
- El **modelo de datos**: entidades del sistema y sus relaciones.
- La **interfaz gráfica**: para tener información de cómo el usuario interactúa con el sistema.
- Las **reglas de negocio**: para entender cuál es el comportamiento esperado del sistema.

El **modelo de datos** se extrae automáticamente a partir del esquema relacional de la BD mediante las técnicas de ingeniería inversa descritas en [8], cuyo motor de ejecución ha sido mejorado y modificado para detectar ocurrencias de patrones de modelos, de modo que el comportamiento de la aplicación pueda ser derivado o supuesto. Así, se puede generar una primera versión del modelo del comportamiento del sistema.

Lo mismo sucede con el modelo de **interfaz gráfica**, considerando que para las entidades principales de un SI se proporcionarán pantallas al menos para las operaciones CRUD.

A partir de estos modelos se derivan los casos de prueba con un enfoque MDT. Las transformaciones de modelo a modelo se implementan en QVT (*query/view/transformation*) [9], un lenguaje estandarizado por OMG (*Object Management Group*) e integrado con UML. Para la generación del código de pruebas es necesario agregar información al modelo que mapee los elementos de la aplicación a los elementos de dicho modelo.

Para este paso utilizaremos un lenguaje de transformación de modelo a texto. Aquí, OMG también ha publicado un lenguaje de transformación llamado MOFM2T [10], que permite la obtención de código fuente o documentación textual a partir de modelos.

Con objeto de conseguir una solución lo más generalista posible utilizaremos, siempre que estén disponibles, especificaciones estándares (propuestas por OMG) para trabajar con la representación de los modelos y las transformaciones. Y, cuando no, aprovecharemos los mecanismos de extensión de UML mediante perfiles, que se basan en el uso de estereotipos y de valores etiquetados.

Para el modelo de pruebas utilizaremos el *UML Testing Profile* (UML-TP) [5] que es el estándar para la representación de pruebas mediante modelos. Para la representación del sistema utilizaremos distintos perfiles para cada aspecto del mismo (por ejemplo, para el modelo de datos utilizamos *UML Data Modeling Profile*, o UDMP [11]).

Finalmente, como motor de ejecución de pruebas utilizamos JUnit [12] y en el caso de pruebas de sistemas Web también utilizamos la herramienta de ejecución de pruebas Selenium [13], por lo que el *framework* generará código para estas plataformas.

3. Propuesta para la generación de casos de prueba

En esta sección describimos los modelos utilizados en nuestra propuesta y los mecanismos de generación de pruebas. Primero se presentan los modelos que representan un SI para poder generar pruebas a partir de ellos. Luego, presentamos los modelos utilizados para las pruebas. Por último se muestra cómo se genera el modelo de pruebas a partir del modelo del sistema, y cómo a

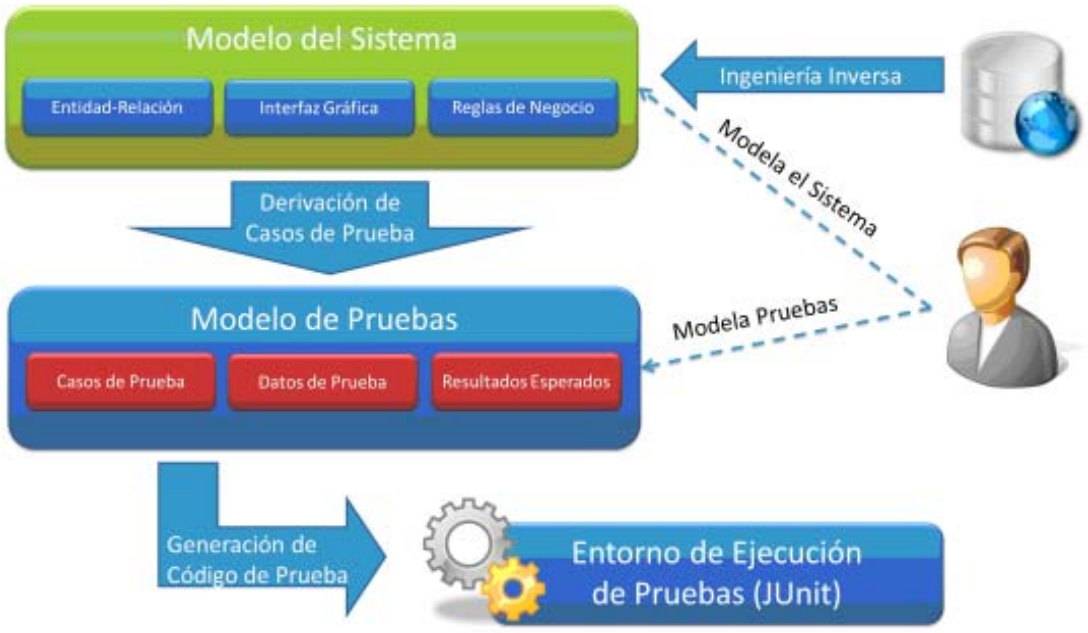

Figura 1. Entorno de diseño de pruebas para sistemas de información.

> “El caso de prueba es el concepto principal en el modelo de prueba, y su comportamiento se describe usando diagramas de comportamiento tales como diagramas de secuencia, máquinas de estado o diagramas de actividad”

partir de estos modelos se genera el código de pruebas.

3.1. Modelo de sistemas de información

El elemento inicial de nuestro proceso es el modelo físico de la base de datos. Otros elementos incluidos en el modelo del sistema tienen como objetivo generar pruebas. Para poder generar pruebas ejecutables sobre una interfaz web, por ejemplo, necesitamos tener información sobre la forma en la que el usuario interactúa con el sistema, pues una prueba automática debería interactuar de la misma forma. Para poder generar datos de prueba y saber si son válidos o no (y así obtener el oráculo del caso de prueba), necesitamos agregar información sobre las reglas de negocio.

A continuación, detallamos cada una de las partes del modelo del sistema. En la **figura 2** se presenta un ejemplo de un modelo de un sistema que maneja facturas y productos, incluyendo cada una de las partes.

Modelo de datos: el modelo de datos se representa con un diagrama de clases UML utilizando el perfil UDMP, que es una extensión de UML propuesta por IBM para diseñar BD en UML que mantiene el poder expresivo del modelo de entidad-relación extendido.

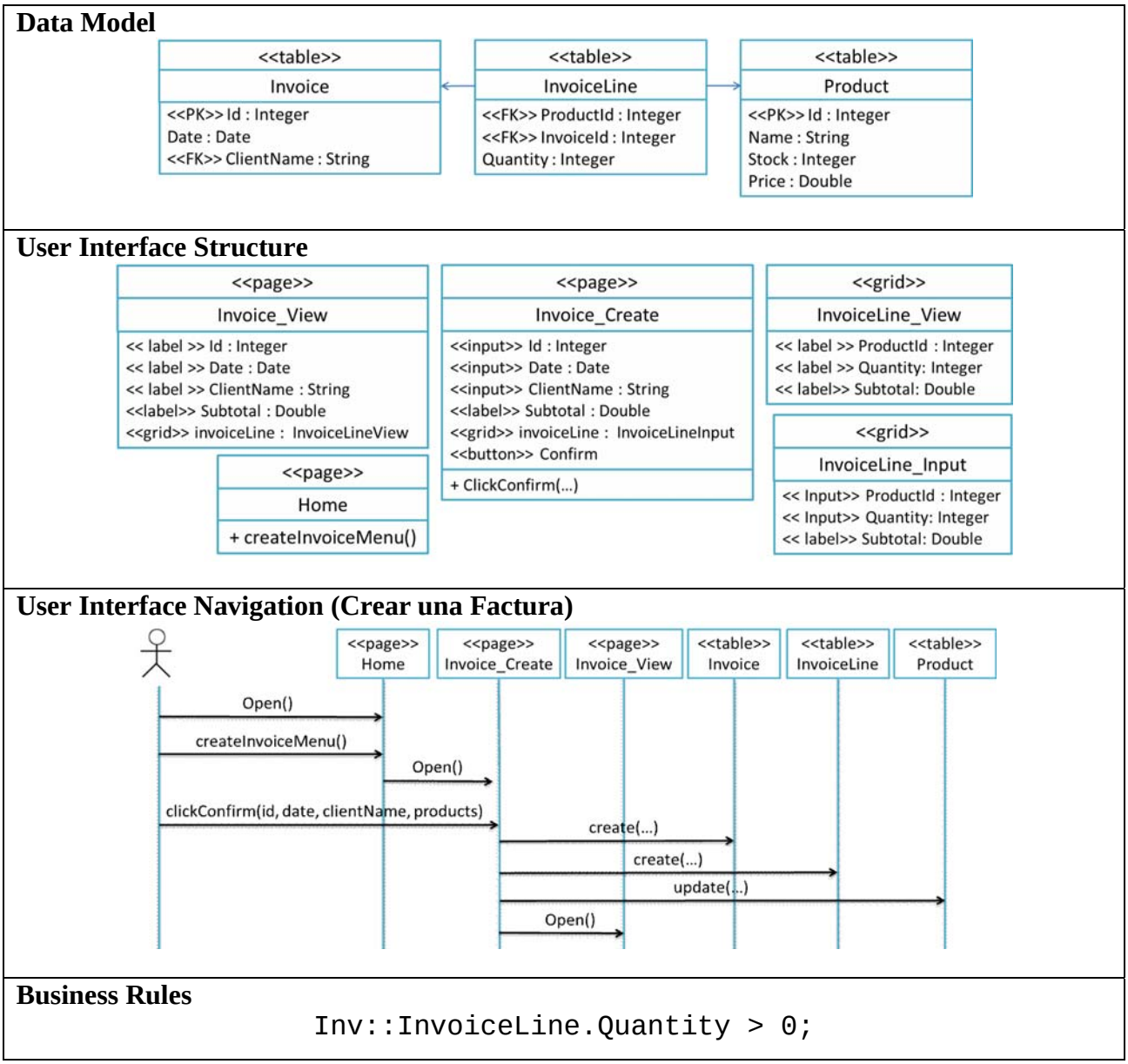

Figura 2. Ejemplo reducido de un modelo del sistema de facturas y productos.

> “Debemos ser capaces de generar casos de prueba que alcancen algún nivel determinado de cobertura, de manera que pueda cuantificarse su completitud”

El UDMP define desde los conceptos a nivel físico y de arquitectura (*Node*, *Tablespace*, *Database*, etc.), hasta los más útiles para el diseño de BD (*Table*, *Column*, *PK*, *FK*, etc.). Existe una correspondencia casi directa entre los estereotipos del perfil UML y los elementos de una BD, siendo completamente independiente del motor de base de datos utilizado. Para cada columna (representada como un atributo de la clase que representa la tabla a la que pertenece) se indica el tipo de datos asociado.

Interfaz de Usuario: el modelo de interfaz de usuario incluye dos tipos de elementos, por un lado la estructura, representada con un diagrama de clases, y por otro lado la navegación, representada con diagramas de comportamiento. En la estructura se modelan los distintos elementos de las pantallas (botones, *inputs*, combos…), con los que interactúa el usuario. Estos serán atributos de las clases, que representan las pantallas. Con el fin de tener información en el modelo para generar pruebas, es necesario mantener también las relaciones existentes entre los elementos de la estructura de la interfaz gráfica y las columnas de la BD. De esta forma podremos saber que la información ingresada por el usuario en determinado campo debe ser almacenada en determinada columna en una tabla de la BD.

Reglas de Negocio: las reglas indican restricciones sobre las variables del sistema, ya sean variables de entrada o sobre columnas de la base de datos, según las restricciones del negocio. Las reglas pueden ser simples (referenciando sólo un atributo o columna) o compuestas (definiendo restricciones sobre varios atributos al mismo tiempo). Por ejemplo, definiendo que el valor de una columna no puede ser nulo o menor que cero, o que uno de los campos debe ser mayor que otro.

Para representar esta información utilizamos un subconjunto del lenguaje OCL.

3.2. Modelo de pruebas

Las pruebas se representan mediante el UML-TP, un lenguaje para diseñar, visualizar, especificar, analizar, construir y documentar artefactos de prueba.

Éste extiende UML con conceptos específicos para las pruebas, agrupándolos en: arquitectura, datos, comportamiento y tiempos de prueba.

La arquitectura de las pruebas contiene la definición de todos los conceptos necesarios para realizar las pruebas. El comportamiento de las pruebas especifica las acciones y evaluaciones necesarias para la prueba. El caso de prueba es el concepto principal en el modelo de prueba, y su comportamiento se describe usando diagramas de comporta-

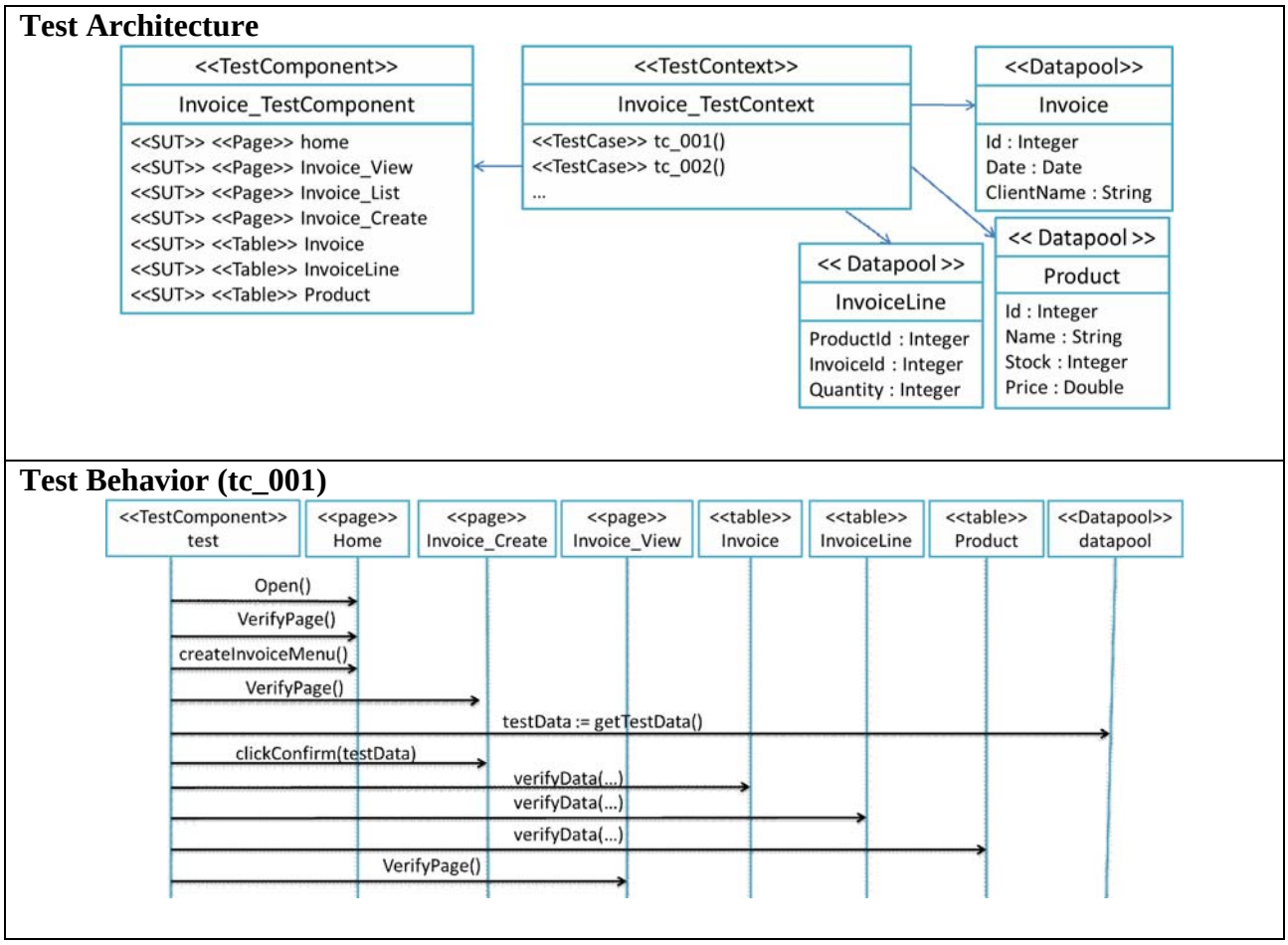

Figura 3. Especificación de pruebas representada con UML-TP.

miento tales como diagramas de secuencia, máquinas de estado o diagramas de actividad.

En este perfil, un caso de prueba (*test case*) es una operación de un contexto de prueba que especifica cómo un conjunto de componentes cooperan con el sistema bajo prueba para alcanzar el objetivo de prueba, retornando un *veredicto* [5].

Por último, y para comprender la forma en que se representan estos casos de prueba de acuerdo con UML-TP, se muestran en la **figura 3** los casos de prueba dentro de la arquitectura de pruebas que, entre otros, incluye:

- Un *Test Context*, conteniendo como métodos los casos de prueba (*test cases*) generados;
- Un *Test Component*, responsable de la inicialización de los casos de prueba y de la interacción con el *System Under Test* (*SUT*);
- Un *datapool* por cada entidad probada; cada *datapool* (conjunto de datos de prueba) tiene un *data selector* por cada *test case* para proveer datos específicos a cada uno.

Como se puede observar, el caso de prueba del ejemplo de la **figura 3** ejercita el escenario presentado en el modelo del sistema, con datos de prueba de acuerdo a las entradas del sistema y del modelo de datos.

Los *datapools* permiten tener un mismo caso de prueba parametrizado con diferentes valores, lo que significa seguir un enfoque de *Data-driven testing* [14]. En la figura, el uso de los *datapools* se muestra simplificado por límites de espacio.

3.3. Generación de los casos de prueba

Los casos de prueba se generan a partir del modelo del sistema, en el que se identifican situaciones de prueba interesantes (obviamente, desde el punto de vista del *testing*) mediante la búsqueda de subestructuras del modelo que concuerden con patrones de modelos predefinidos (una situación de prueba interesante, por ejemplo, podría ser la existencia de una asociación *1:** entre dos clases: quizás sea interesante probar el funcionamiento del SI ante la inserción de un registro en la tabla secundaria sin un registro asociado en la tabla principal).

Además, y con el fin de conocer su completitud y calidad, los casos de prueba se construyen de acuerdo a criterios de cobertura definidos para modelos.

Es importante destacar que las pruebas generadas atacan el sistema completo, basándose en las estructuras de la BD, por lo que se está verificando que el sistema bajo pruebas sea capaz de manejarlas correctamente. Claramente, el ejemplo presentado será bien manejado por el gestor de BD, pero es muy importante ver que el sistema reaccione adecuadamente ante la excepción que le dará la ejecución de la sentencia SQL, y que el modelo de datos de la BD continúe consistente con los modelos de las capas superiores.

La información antes presentada del SI es útil para derivar *test cases*. De hecho, es un buen punto de inicio para aplicar diversos criterios de cobertura.

Toda estrategia de generación de pruebas debe ser mensurable. En nuestro caso, también debemos ser capaces de generar casos de prueba que alcancen algún nivel determinado de cobertura, de manera que pueda cuantificarse su completitud. Los criterios son un mecanismo para comprobar, validar y cuantificar que lo que estamos probando realmente "cubre" el modelo, de acuerdo a una teoría de errores dada.

Veamos esto considerando que a partir del modelo del sistema generamos dos tipos de pruebas diferentes:

- Pruebas de operaciones atómicas: para funcionalidades específicas como la creación, actualización o borrado de instancias.
- Pruebas del ciclo de vida de entidades: basadas en el ciclo de vida de una entidad se combinan las operaciones de creación, lectura, actualización y borrado de las entidades.

Para la **prueba de las operaciones atómicas** se diseñan datos de prueba de acuerdo a las entradas del usuario (*inputs* en las pantallas correspondientes) y a la subestructura del modelo de datos asociado, y claro, a la relación entre ambos modelos (cómo una entrada de usuario es almacenada en el modelo de datos).

Si esta información la complementamos con las reglas de negocio seremos capaces de aplicar diferentes criterios de cobertura de datos tales como partición de equivalencia, valores límites [15] y luego combinación de datos por pares , pudiendo identificar qué clases de equivalencia son válidas y cuáles inválidas, y de esa forma conociendo cuál es el resultado esperado (si se usan datos inválidos el resultado es que la operación falle). Además, adaptamos ciertos criterios de cobertura para diagramas de clase propuestos por Andrews et al. [17] para el modelo de datos [18]; estos son: **AEM** (*Association-end multiplicity*): requiriendo que se pruebe cada par de multiplicidades representativo en las asociaciones del modelo (si existe una asociación cuya multiplicidad es, en un extremo, *p..n*, debería instanciarse la asociación con *p* elementos, *n* elementos y con uno o más valores en el intervalo; **CA** (*Class attribute*): requiriendo que se pruebe con conjuntos de valores representativos de los diferentes atributos de cada clase.

Para la **prueba del ciclo de vida de las entidades** hemos adaptado los criterios de cobertura de máquinas de estado [6] (todos los estados, todas las transiciones, combinaciones de entrada/salida para cada estado) dado que el ciclo de vida de una entidad puede ser descrito con una máquina de estados considerando las operaciones CRUD, que afectan su estado.

El ciclo de vida de un dato viene dado por una ocurrencia de una expresión regular del tipo $C \cdot R \cdot (U_i \cdot R)^* \cdot D \cdot R$ [14], en donde cada U_i representa una de las múltiples operaciones de actualización del objeto. Las operaciones *R (Read)* son simples operaciones de lectura que no alteran el estado del objeto ni el de sus datos persistentes asociados, por lo que se utilizan para construir el oráculo tras cada operación que sí suponga cambio de estado.

3.4. Generación del código de pruebas

El modelo de prueba se genera mediante transformaciones realizadas sobre el modelo del sistema, y luego con el modelo de prueba se genera código de prueba.

Entonces, a través de transformaciones aplicadas al modelo del sistema, se generan pruebas en el modelo de pruebas considerando los criterios de cobertura mencionados. Este modelo de pruebas (representado con UML-TP) se transforma finalmente a código de prueba ejecutable con JUnit y Selenium sobre la interfaz web, para poder analizar la calidad del sistema con respecto a la forma en que maneja sus estructuras de datos.

Selenium es tal vez una de las herramientas *opensource* más populares para la prueba de aplicaciones web: es una herramienta con un enfoque *record & playback* [19] (captura las acciones de una ejecución manual de un caso de prueba, las guarda como una secuencia de comandos en un *script* y luego puede ejecutar ese *script* simulando así las acciones del usuario).

Selenium no permite parametrizar datos (para seguir un enfoque de *Data-driven testing*) en forma nativa, pero en cambio ofrece una muy buena integración con JUnit que a partir de la versión 4 contempla la parametrización de pruebas.

Existen en Selenium distintos comandos de acuerdo al tipo de elemento con el que se está interactuando (por ejemplo, el comando "`type`" para ingresar valores en elementos tipo "`input`", o el comando "`click`" para activar elementos tipo "`button`").

Estos comandos hacen referencia al elemento HTML sobre el que se está actuando, ya sea un *input* o un botón, se deberá contar con una referencia al mismo. Por este motivo, para la generación automática de pruebas ejecutables, también es necesario modelar un mapeo entre los elementos del modelo de interfaz gráfica y los elementos HTML correspondientes.

Para esto se sigue el enfoque *Model-to-Implementation Matching* propuesto por Xu [20], donde se debe agregar esta información (ya sea en forma manual, o asistida también con técnicas de ingeniería inversa sobre las páginas web).

```
@RunWith(value = Parameterized.class)
public class invoice_test {
    private String id;
    private String date;
    private String clientName;
    private String subtotal;
    private ArrayList<String> lines = new ArrayList<String>();
    public invoice_test(String id, String date, String clientName,
                String subtotal, String... invoiceLine) {
        this.id = id; this.date = date;
        this.clientName = clientName; this.subtotal = subtotal;
        for (String s : invoiceLine) {
          this.lines.add(s);
        }
    }
   @Parameters
   public static Collection<Object[]> data() throws Exception {
     invoice_datapool dp = new invoice_datapool();
     return dp.getData(); //reads a file with the test data
   }
   I_home home;
   I_invoice_create invoice_create;
   I_invoice_view invoice_view;
   I_database database;
   @Before
   public void setUp() throws Exception {
        …
   }
   @Test
   public void tc001_invoice() throws Exception {
        home.open();
        home.VerifyPage();
        home.CreateInvoiceMenu();
        invoice_create.VerifyPage();
        invoice_create.ClickConfirm(id,date,clientName,subtotal,lines);

     database.VerifyInvoiceDataExists(id,date,clientName,subtotal,lines)
;
   }
}
```

```
public class invoice_create_selenium {
    public void ClickConfirm(String id, String date, String clientName,
        String subtotal, ArrayList<invoiceLine> invoiceLine) throws
Exception{
        selenium.type("id=span_INVOICEID", id);
        selenium.type("id=span_INVOICEDATE", date);
        selenium.type("id=span_CLIENTID", clientName);
```

Figura 4. Código de pruebas con JUnit y Selenium.

En la **figura 4** se muestra un fragmento de código de pruebas utilizando JUnit y Selenium, basados en las pruebas especificadas en la **figura 3**.

Está diseñado de forma tal que la prueba JUnit sigue la especificación de pasos de la prueba, los cuales se ejecutan con Selenium en un conjunto de *wrappers* que podría incluso ser reemplazado por otro si se quiere cambiar Selenium por otra plataforma de ejecución (se incluye en la figura un *test case* en JUnit y uno de los *wrappers* que contiene un método *ClickConfirm* utilizado en la prueba). Además, se aprovecha el mecanismo de parametrización de pruebas que brinda JUnit desde su versión 4, tomando los datos de prueba desde un archivo.

4. Trabajos relacionados

Existen varios trabajos que generan pruebas y datos de prueba para sistemas con BD, pero ninguno toma el enfoque dirigido por modelos considerando las operaciones básicas de un SI como proponemos hacer nosotros.

Algunos, por ejemplo, extienden la cobertura basada en líneas de código considerando las particularidades de la interacción del sistema bajo prueba con la BD [21][22][23], así como las distintas condiciones que se puedan plantear sobre las SQLs que se ejecutan [24][25]. Así, se generan las instancias de la BD necesarias para cubrir estas situaciones.

Tanto en [26] como en [27] se plantea especificar de alguna forma los resultados de las SQL implicadas en la prueba, para de esa forma generar datos para poblar la BD. La propuesta planteada en [28] toma como entradas el esquema de la BD y datos de prueba categorizados dados por el usuario, con lo que genera casos de prueba y estados iniciales para la BD, validando tanto las salidas del sistema como el estado final de la BD.

En [29] se generan estados de la BD de acuerdo a las restricciones de integridad del esquema relacional: primero se generan reglas de consistencia de acuerdo a las claves y referencias foráneas, y luego se generan datos que cumplan con ellas.

Existen muchos trabajos que generan pruebas automáticamente a partir de modelos UML [30][31] pero, hasta donde conocemos, solo [32] considera el caso especial de los SI con BD (los cuales tienen particularidades muy importantes a ser tenidas en cuenta).

En esa propuesta, Fujiwara et al. proponen generar pruebas considerando un diagrama

de clases para representar el modelo de datos, y otro para representar las pantallas. Las relaciones entre tablas son representadas con restricciones OCL, así como también las pre y post condiciones de los métodos a probar. Todo el modelo que especifica el sistema debe ser proporcionado manualmente, y por lo tanto luego debe ser mantenido. Las pruebas que generan están centradas en las restricciones dadas, mientras que en nuestra propuesta prestamos especial atención al modelo de datos, y la especificación del sistema es obtenida semiautomáticamente, reduciendo los costos de mantenimiento.

5. Conclusiones y trabajo futuro

Este artículo ha presentado un nuevo enfoque para probar SI que manejan BD, en el que se pone especial atención en cubrir todas las entidades encontradas en el modelo de datos. Este enfoque tiene en cuenta que en estos sistemas lo importante es que la información esté correctamente almacenada, por lo que se prueban las operaciones que operan sobre estas estructuras conforme a diversos criterios de cobertura.

El entorno de generación de pruebas es dirigido por modelos y está casi completamente basado en estándares (UML, UML-TP, UDMP, QVT, MOFM2T), por lo que puede ser adoptado con casi cualquier herramienta que dé soporte a UML. Por este motivo, casi no se necesita implementación de nuevas herramientas para dar soporte a esta metodología.

Gracias a la aplicación de lata metodología proporcionada, somos capaces de generar pruebas que nos ayudan a garantizar la calidad de los sistemas de información que desarrollamos.

En este momento estamos trabajando en extender la propuesta con mayores criterios de cobertura, a modo de generar mejores casos de prueba.

Por otra parte, el modelo del sistema actualmente representa características funcionales del sistema. Este modelo lo queremos extender para poder representar características no funcionales, tales como el rendimiento, y de esa forma tener información suficiente para generar escenarios de prueba que verifiquen este otro tipo de requisitos.

Agradecimientos

Este trabajo ha sido parcialmente financiado por la Agencia Nacional de Investigación e Innovación (ANII, Uruguay), por el proyecto GEODAS-BC (Ministerio de Economía y Competitividad y Fondo Europeo de Desarrollo Regional FEDER, TIN2012-37493-C03-01).

Referencias

[1] J.-L. Hainaut, V. Englebert, J. Henrard, J.-M. Hick, D. Roland. "Database reverse engineering: From requirements to CARE tools". *Autom. Softw. Eng.*, vol. 3, no. 1–2, pp. 9–45, 1996.

[2] M. H. Alalfi, J. R. Cordy, T. R. Dean. "SQL2XMI: Reverse Engineering of UML-ER Diagrams from Relational Database Schemas". *Working Conference on Reverse Engineering, 1448047*, 2008, pp. 187–191.

[3] M. Blaha. "A retrospective on industrial database reverse engineering projects - Part 1". *Eighth Working Conference on Reverse Engineering 2001*, pp. 136–146. <http://www.researchgate.net/publication/3919794_A_retrospective_on_industrial_database_reverse_engineering projects._1>.

[4] I. García Rodríguez de Guzmán. "*Pressweb: un proceso para la reingeniería de sistemas heredados hacia servicios web*". UCLM, 2007.

[5] P. Baker, Z. R. Dai, J. Grabowski, O. Haugen, I. Schieferdecker, C. Williams. *Model-Driven Testing: Using the UML Testing Profile*. Springer-Verlag New York, Inc., 2007.

[6] J. Offutt, S. Liu, A. Abdurazik, P. Ammann. "Generating test data from state•]based specifications". *Softw. Test. Verification Reliab., vol. 13*, pp. 25–53, 2003.

[7] M. Utting, B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2010.

[8] M. Polo, I. García-Rodríguez, M. Piattini. "An MDA-based approach for database re-engineering". *J. Softw. Maint. Evol. Res. Pr., vol. 19*, pp. 383–417, 2007.

[9] OMG. "*Meta Object Facility 2.0 Query/View/Transformation Specification*", 2005.

[10] OMG. "*MOF Model to Text Transformation Language (MOFM2T), 1.0*", 2008.

[11] D. Gornik. "*UML Data Modeling Profile*". IBM, Rational Software, 2002.

[12] E. G. Kent Beck. "*JUnit*". 1997. Disponible en <http://www.junit.org>.

[**13] J. Huggins.** "*Selenium*". 2004. Disponible en <http://seleniumhq.org/>. Último acceso: 06-Mar-2013.

[14] T. Koomen, L. van der Aalst, B. Broekman, M. Vroon. TMap Next, for result-driven testing. UTN Publishers, 2006.

[15] G. Myers. *The Art of Software Testing*. John Wiley & Son. Inc., 2004.

[16] M. P. Usaola, B. P. Lamancha. "*A framework and a web implementation for combinatorial testing*". Alarcos Research Group White Paper, 2010. <http://alarcosj.esi.uclm.es/CombTestWeb/stuff/wpCombinatorial.pdf>. Último acceso: 30-octubre-2013.

[17] A. Andrews, R. France, S. Ghosh, G. Craig. "Test adequacy criteria for UML design models". *Softw. Test. Verification Reliab., vol. 13*, pp. 95–127, 2003.

[18] F. Toledo, B. P. Lamancha, M. P. Usaola. "Data model centered test case desing. A model-driven approach". *The Fourth International Conference on Advances in System Testing and Validation Lifecycle* (VALID), 2012.

[19] D. Graham, M. Fewster. *Experiences of Test Automation: Case Studies of Software Test Automation*. Addison-Wesley Professional, 2012. ISBN-10: 0321754069.

[20] D. Xu. "A tool for automated test code generation from high-level Petri nets". *Appl. Theory Petri Nets*, pp. 308–317, 2011.

[21] K. Haller. "White-box testing for database-driven applications: A requirements analysis". *DBTest '09 Proceedings of the Second International Workshop on Testing Database Systems*, p. 13, 2009.

[22] S. Shelar, Sdsawarkar. "Database instances generation tool for white-box testing". *2009 International Conference on Signal Acquisition and Processing*, pp. 112–116.

[23] M. Emmi, R. Majumdar, K. Sen. "Dynamic test input generation for database applications". *ISSTA'07: Software Testing and Analysis*, 2007, pp. 151–162.

[24] J. Tuya, M. J. Suárez-Cabal, C. De La Riva. "Full predicate coverage for testing SQL database queries". *Softw. Test. Verification Reliab.*, vol. 20, pp. 237–288, 2010.

[25] M. J. Suárez-Cabal, C. De La Riva, J. Tuya. "Populating test databases for testing SQL queries". *Ieee Lat. Am. Trans., vol. 8*, pp. 164–171, 2010.

[26] C. Binnig, D. Kossmann, E. Lo. "Multi-RQP - generating test databases for the functional testing of OLTP applications". *DBtest'08: International Workshop on Testing Database Systems*, 2008, p. 5.

[27] A. Arasu, R. Kaushik, J. Li. "Data generation using declarative constraints". *International conference on Management of data*, 2011, pp. 685–696.

[28] D. Chays, Y. Deng. "Demonstration of AGENDA tool set for testing relational data-base applications". *Proceedings of the 25th International Conference on Software Engineering*, pp. 802–803, 2003. <http://ase-conferences.org/ase/past/ase2003/demos/yuetang.pdf>.

[29] A. Neufeld, G. Moerkotte, P. C. Loekemann. "Generating consistent test data: Re-stricting the search space by a generator formula". *Vldb J., vol. 2*, pp. 173–213, 1993.

[30] A. Brucker, M. Krieger, D. Longuet, B. Wolff. "A specification-based test case gen-eration method for UML/OCL". *Models Softw. Eng.*, pp. 334–348, 2011.

[31] J. Offutt, A. Abdurazik. "Generating tests from UML specifications". *Second International Conference on the Unified Modeling Language (UML99)*, 1999.

[32] S. Fujiwara, K. Munakata, Y. Maeda, A. Katayama, T. Uehara. "Test data generation for web application using a UML class diagram with OCL constraints". *Innov. Syst. Softw. Eng.*, pp. 1–8, 2011.

Javier Tuya
Coordinador del grupo de trabajo AEN/CTN71/SC7/GT26 de AENOR; Catedrático de la Universidad de Oviedo

<tuya@uniovi.es>

La norma ISO/IEC/IEEE 29119 - *Software Testing*

Resumen: *El autor del artículo, como coordinador del grupo de trabajo AEN/CTN71/SC7/GT26 de AENOR, repasa la estructura, contenidos principales e interrelaciones del éstandar ISO/IEC/IEEE 29119* ("The International Software Testing Standard") *en cuya elaboración, iniciada en 2007, su grupo ha venido participando.*

Palabras clave: *AENOR, evaluación de procesos, ISO/IEC/IEEE 29119, ISO/IEC 33063.3, normativa, pruebas de software.*

Autor

Pablo Javier Tuya González es Catedrático de Universidad y director del Grupo de Investigación en Ingeniería del Software (GIIS, <http://giis.uniovi.es/>) de la Universidad de Oviedo. Actualmente es director de la Cátedra Indra-Uniovi y coordinador del grupo de trabajo de AENOR AEN/CTN 71/SC7/GT26 Pruebas del Software. Sus líneas de investigación se centran en las pruebas del software, en especial para aplicaciones con bases de datos y orientadas a servicios. Su página personal se encuentra en <http://www.di.uniovi.es/~tuya/>.

La elaboración de la nueva serie de normas sobre pruebas de software recientemente publicadas por ISO comenzó su andadura en 2007, año en el que ISO aprobó la constitución del grupo de trabajo WG26 dentro del subcomité ISO/IEC JTC1/SC7 "*Software and Systems Engineering*".

Posteriormente, AENOR constituyó el correspondiente grupo de trabajo AEN/CTN71/SC7/GT26 – "Pruebas de Software" <http://in2test.lsi.uniovi.es/gt26/> para trabajar en la elaboración de la norma. Este grupo ha contribuido intensamente en su elaboración mediante la participación en las reuniones internacionales del grupo WG26 y en las actividades de revisión, propuestas de cambio y votaciones en las diferentes fases por las que han pasado las diferentes partes de la norma.

El objetivo es producir un conjunto de normas sobre pruebas de software aplicables a todos los tipos de productos basados en software, proporcionando un tratamiento consistente y unificado, e integrando el fragmentado cuerpo normativo actual relativo a las pruebas de software ofrecido por BSI, IEEE e ISO/IEC.

Esta serie de normas, denominada ISO/IEC/IEEE 29119 consta de 4 partes. Las tres primeras han sido publicadas en agosto de 2013. La cuarta parte está en progreso:
1) *ISO/IEC/IEEE 29119-1:2013 Software and systems engineering - Software testing - Part 1: Concepts and definitions.*
2) *ISO/IEC/IEEE 29119-2:2013 Software and systems engineering - Software testing - Part 2: Test processes.*
3) *ISO/IEC/IEEE 29119-3:2013 Software and systems engineering - Software testing - Part 3: Test documentation.*
4) *ISO/IEC/IEEE DIS 29119-4 Software and systems engineering - Software testing - Part 4: Test techniques.*

La parte 1 (Conceptos y definiciones) facilita el uso de las otras, proporcionando definiciones de términos sobre pruebas y discusión de conceptos clave para la comprensión del resto de normas, así como diferentes formas de aplicar los procesos definidos en la parte 2.

La parte 2 (Procesos de pruebas) especifica los procesos de pruebas que pueden ser usados para gobernar, gestionar e implementar pruebas de software para cualquier organización, proyecto o actividad de pruebas. Para ello incluye un conjunto de descripciones genéricas de los procesos.

La parte 3 (Documentación de pruebas) especifica plantillas de documentación que pueden ser usadas por las organizaciones, proyectos o actividades. Describe la documentación que es producida como salida de los procesos especificados en la parte 2.

La parte 4 (Técnicas de prueba) define un conjunto de técnicas de diseño de pruebas que pueden ser usadas durante los procesos de diseño e implementación de la parte 2.

Además de lo anterior, en 2012 se aprobó iniciar una quinta parte sobre "*Keyword Driven Testing*".

Otra norma íntimamente relacionada con la serie 29119, gestionada desde grupo de trabajo WG10, y que está actualmente en elaboración es *ISO/IEC 33063.3 Process assessment - An exemplar process assessment model for software testing.* Suministra un Modelo de Evaluación de Procesos de pruebas de software para uso en la realización de evaluaciones de conformidad de acuerdo con los requisitos de *ISO/IEC 33002 – Process Assessment – Requirements for performing process assessments.* Para ello utiliza como Modelo de Referencia de Procesos el definido en la parte 2 de la serie 29119.

Un marco metodológico para evaluar técnicas y herramientas para pruebas del software

Tanja E. J. Vos[1], Beatriz Marín[2], María José Escalona Cuaresma[3]

[1]Centro de Métodos de Producción de Software, Universidad Politécnica de Valencia (España); [2]Universidad Diego Portales, Santiago (Chile); [3]Ingeniería Web y Testing Temprano (IWT2), Universidad de Sevilla (España)

<tvos@pros.upv.es>, <beatriz.marin@mail.udp.cl>, <mjescalona@us.es>

Resumen: *Actualmente existe una necesidad real en el sector industrial de disponer de conocimiento para decidir qué técnicas de pruebas deben usarse según los objetivos de pruebas, y para conocer cómo de usables (efectivas, eficientes y satisfactorias) pueden llegar a ser estas técnicas. Sin embargo, estas guías en realidad no existen. Podríamos plantearnos como medio para conseguirlas el realizar estudios comparativos de evaluación de técnicas de pruebas y herramientas basadas en casos de estudio. Sin embargo, estos estudios también son poco viables por falta de disponibilidad de dichos casos. En este trabajo, daremos un primer paso para crear una primera aproximación a un marco de trabajo de evaluación que permita simplificar el diseño de los casos de estudio a comparar en las herramientas de pruebas de software, haciendo los resultados lo más precisos, legibles y fáciles de comparar.*

Palabras clave: *Guías, Ingeniería del Software, pruebas del software, organizaciones, taxonomía.*

Autoras

Tanja E. J. Vos es profesora en la Universidad Politecnica de Valencia, donde además es responsable del grupo STaQ (*Software Testing and Quality*) del centro PROS, Métodos de Producción de Software. Ha participado tanto en proyectos de investigación para el gobierno como en colaboración con la industria. Es licenciada en Informática y realizó su doctorado en verificación de sistemas distribuidos también por la Universidad de Utrecht. Tanja también ha trabajado como investigadora y profesora en la Universidad de Utrecht (Países Bajos), Universidad Mayor de San Simón (Bolivia), Universidad de Cambridge (Inglaterra) y Universidad Mediterránea de Ciencia y Tecnología (España).

Beatriz Marín es profesora en la Universidad Diego Portales, Chile. Es Doctora en Informática por la Universidad Politécnica de Valencia, España. Ha publicado artículos científicos en revistas y conferencias internacionales de reconocido prestigio (DKE, TOSEM, ESEM, QSIC, etc.). Sus principales líneas de investigación son la Ingeniería de Software, Pruebas del Software, Desarrollo de Software Dirigido por Modelos, Medición de Software, e Ingeniería de Software Empírica.

María José Escalona Cuaresma es doctora en Ingeniería Informática por la Universidad de Sevilla . Actualmente trabaja como profesora titular en la ETS de Ingeniería Informática de la misma Universidad donde, además, ostenta el cargo de Subdirectora de Extensión Universitaria y Relaciones Internacionales. Es la directora del grupo de Investigación de Ingeniería Web y Testing Temprano <http://www.iwt2.org/> y ha dirigido y participado en un gran número de proyectos tanto de investigación como de transferencia tecnológica. Entre sus líneas de investigación destacan la Ingeniería de Requisitos, la Ingeniería Web, el *testing* temprano y la Calidad del software. Es además miembro del comité de programa de relevantes congresos en Ingeniería del Software y participa como revisora en un gran número de revistas relacionadas con sus áreas de investigación.

1. Introducción

Para realizar las pruebas de software se necesita tomar decisiones informadas acerca de qué técnicas usar en situaciones específicas, y estimar el tiempo y esfuerzo necesarios para aplicarlas. Sin embargo, actualmente no es posible encontrar guías claras para estas tareas debido principalmente a la falta de estudios empíricos que sirvan como experiencias documentadas para estudios secundarios de ingeniería de software basada en evidencia (en inglés *Evidence-Based Software Engineering* – EBSE) [1].

En este contexto, es necesario contar con más estudios que evalúen y comparen empíricamente técnicas y herramientas para realizar pruebas de software [2][3].

Sin embargo, para asegurar que la evidencia de estos estudios pueda resultar en guías adecuadas, los casos de estudios evaluados deberían:

- Involucrar sistemas y sujetos realistas y no programas de ejemplo y estudiantes, como es el caso de la mayoría de estudios realizados.
- Ser realizados con rigurosidad para asegurar que cualquier beneficio identificado durante el estudio es claramente derivado de la técnica de pruebas estudiada.
- Asegurar que diferentes estudios pueden ser comparados.

Aunque este tipo de investigación es caro, difícil, y consume demasiado tiempo, es fundamental [4].

Desafortunadamente, las empresas a menudo se muestran reacias a participar en casos de estudio. Algunas no están dispuestas a probar nuevos enfoques que quizás no han sido anteriormente ensayados. Otras se preocupan de que se podrían revelar fallas críticas, métricas pobres o bajo rendimiento. Otras empresas no están dispuestas a permitir a los investigadores realizar los estudios por temor a perder información confidencial, que ellos hagan más lento el trabajo del equipo, o simplemente no sepan cómo hacerlo. Independientemente de los motivos, es necesario superar estas barreras con el fin de avanzar y crear el cuerpo de evidencia necesario.

En este trabajo se propone un marco *metodológico* general para reducir algunas de las barreras de entrada para realizar casos de estudio. Este marco metodológico simplificará el diseño de los casos de estudio para comparar técnicas de pruebas de software, asegurando que la mayoría de las guías definidas para realizar estudios empíricos han sido seguidas. Además, si los casos de estudio son ejecutados con un diseño similar (es decir, por la instanciación del marco propuesto), será posible replicar los estudios, y con la evidencia generada, será más fácil comparar y efectivamente agregar estos estudios en nuevos estudios secundarios.

El marco que se presenta en este trabajo ha evolucionado durante los últimos años mediante la realización de casos de estudio realizados para evaluar técnicas de pruebas [5]. La necesidad de contar con un marco como el que se describe en este trabajo ha surgido hace algunos años durante la ejecución del proyecto EvoTest (IST-33472, 2007-2009, ver [6]) y continuado durante la ejecución del proyecto FITTEST (ICT-257574, 2010-2013, ver [7]).

Ambos proyectos tienen como objetivo el desarrollo de herramientas de pruebas que durante el proyecto deben ser evaluadas en

> "En este trabajo se propone un marco *metodológico* general para reducir algunas de las barreras de entrada para realizar casos de estudio"

entornos industriales. En estos proyectos, se necesita un marco que pueda ser instanciado para cualquier tratamiento, sujeto y objeto y que simplifique el diseño de los estudios evaluativos sugiriendo preguntas y medidas relevantes.

Dado que este marco no existe, en este trabajo se presenta un marco asegurándonos de seguir las guías y directrices encontradas en la literatura. Hasta la fecha, el marco se ha utilizado exitosamente en varios casos de estudio de los proyectos EvoTest y FITTEST.

2. Trabajo relacionado

La mayoría del trabajo relacionado existente presenta marcos *organizacionales* y guías, como por ejemplo listas de pasos que deben ser llevados a cabo, o advertencias de que los estudios deben ser diseñados cuidadosamente y de que los factores que lleven a confusión deben ser minimizados. Con la excepción de [8], que se limita a la inyección de fallos, no se ha encontrado ningún trabajo que especifique cómo evaluar técnicas de pruebas de software, cómo deben ser definidas las preguntas de investigación, qué variables deben ser medidas, cuáles pueden ser las amenazas a la validez de los estudios, etc.

Lott y Rombach [9] describen un esquema de caracterización para experimentos de pruebas de software. Este esquema es similar al esquema general presentado en [10], pero adaptado para evaluar técnicas de prueba. Este esquema ayuda a los usuarios a organizar y evaluar el diseño de un experimento. Sin embargo, no ofrece ninguna ayuda concreta sobre la forma en que los distintos componentes de un experimento pueden ser diseñados y qué puede ser medido.

Do et al. [11] definen SIR, un Repositorio de Artefactos de Infraestructura de Software (<http://sir.unl.edu/>), para apoyar experimentos controlados con técnicas de pruebas de software. La principal contribución de su trabajo es un conjunto de programas que pueden ser utilizados para evaluar las pruebas técnicas. No se presentan directrices metodológicas claras.

El trabajo de Eldh et. al [8] describe un marco para la comparación de técnicas de prueba basadas en inyección de fallos. Los pasos del marco son: preparar muestras de código con fallos conocidos mediante inyección de fallos; seleccionar una técnica de pruebas; realizar el experimento y recopilar datos; analizar los datos y repetir el experimento si es necesario. Este trabajo describe los desafíos industriales para cada paso y detalla un marco metodológico interesante, pero se limita a inyección de fallos.

El marco organizacional DESMET [12], al igual que el marco presentado en este trabajo, ha sido especialmente desarrollado para evaluar métodos y herramientas dentro de las empresas. Y no está especialmente dirigido a investigadores sino a los vendedores de herramientas, ingenieros de software que deseen evaluar una propuesta de modificación, etc.

3. Un marco para evaluar herramientas de pruebas

Imagine que la empresa C quiere evaluar una técnica o herramienta de pruebas T para ver si es usable y vale la pena incorporar esta técnica o herramienta en sus procesos de pruebas. Las siguientes secciones ayudan a definir un caso de estudio.

3.1 Objetivo del estudio - ¿Qué alcanzar?

El marco general se centra en efectividad, eficiencia y satisfacción subjetiva. De esta manera, las preguntas de investigación para cada caso de estudio corresponden a instancias de:

RQ_1 - ¿Cómo contribuye T a la efectividad de las pruebas (capacidades de encontrar fallos) cuando es usada en entornos de pruebas reales de C y comparada con las actuales prácticas de pruebas usadas en C?

RQ_2 -¿Cómo contribuye T a la eficiencia de las pruebas cuando es usada en entornos de pruebas reales de C y comparada con las actuales prácticas de pruebas usadas en C?

RQ_3 - ¿Qué tan satisfechos están los profesionales de pruebas de C durante el aprendizaje, instalación, configuración, y uso de T cuando es usado en entornos de pruebas reales?

3.2 Sujetos - ¿Quién aplica las técnicas/herramientas?

Idealmente, los sujetos son trabajadores de C. Éstos deberían ser las personas que normalmente utilizan las técnicas o herramientas que están siendo comparadas con T. Si por alguna razón esto no es posible (por ejemplo, debido a la falta de tiempo y recursos, la herramienta es un prototipo académico, etc.), entonces investigadores o desarrolladores de herramientas pueden evaluar las mismas. Sin embargo, esto significaría que la satisfacción subjetiva no puede ser medida y no se pueden obtener resultados de las capacidades de la herramienta dentro de un entorno industrial.

3.3. Objetos - ¿Cuáles son los proyectos piloto?

El sistema sometido a pruebas debería ser un sistema que es típico de C (es decir, manteniendo la manera en que el software es desarrollado, la forma en que se prueba, idiomas utilizados, etc.). Además, la información disponible de este sistema debería ser determinada o medida para hacer la comparación con T. Para ello, las siguientes preguntas deben ser respondidas:

S1. ¿Se tendrá acceso a un sistema con fallos conocidos? ¿Qué información se tiene acerca de esos fallos?

S2. ¿Se puede inyectar fallos al sistema?

S3. ¿C recopila datos de sus proyectos como práctica estándar? ¿Qué datos son éstos? ¿Están estos datos disponibles para la comparación? ¿Se tiene una línea base de la empresa? ¿Se tiene acceso a proyectos similares?

S4. ¿La empresa C tiene suficiente tiempo y recursos para realizar varias rondas de pruebas? O de manera más concreta:

- ¿Está la empresa C dispuesta a hacer una nueva *testsuite* TS_N con alguna técnica/herramienta T_{known} ya utilizada en la empresa o T_{unkown} que es nueva en la empresa?
- ¿Se puede utilizar para comparar una *testsuite* TS_C existente? ¿Se conocen las técnicas que fueron utilizadas para crear esa *testsuite*? Y ¿cuánto tiempo tomó su desarrollo?

3.4. Variables - ¿Qué datos recolectar?

Las variables independientes consideradas son: método de pruebas T usado; complejidad de los sistemas industriales, nivel de experiencia de los ingenieros de C que van a hacer las pruebas.

La variable dependiente está orientada a medir eficacia, eficiencia y satisfacción. La siguiente es una lista de las mediciones que se pueden realizar. Para una instanciación específica de este marco en una empresa, algunas de estas variables podrían no ser aplicables.

1. Midiendo Efectividad

a) Número de casos de prueba diseñados o generados.

b) Número de casos de prueba inválidos generados.

c) Número de casos de prueba generados repetidos.

> “El marco organizacional DESMET, al igual que el marco presentado en este trabajo, ha sido especialmente desarrollado para evaluar métodos y herramientas dentro de las empresas”

d) Número de deficiencias observadas.
e) Número de fallos encontrados.
f) Número de falsos positivos.
g) Número de falsos negativos.
h) Tipo y causa de los fallos que fueron encontrados.
i) Cobertura alcanzada (estimada o medida).

2. Midiendo Eficiencia
j) Tiempo necesitado para aprender el método de pruebas T.
k) Tiempo necesitado para diseñar o generar los casos de pruebas.
l) Tiempo necesitado para establecer la infraestructura de pruebas específica para T (instalar, configurar, desarrollar controladores de pruebas, etc.).
m) Tiempo necesitado para probar y observar deficiencias (es decir, planificación, implementación, y ejecución).
n) Tiempo necesitado para identificar tipos de fallos y causas para cada deficiencia observada.

3. Midiendo Satisfacción Subjetiva
o) Cuestionario de Puntuación de Usabilidad del Sistema (*System Usability Score* – SUS por sus siglas en inglés) [13] consistente de 10 preguntas con escala Likert de 5 puntos y un puntaje total.
p) 5 reacciones (a través de cartas de reacción) que serán usadas para crear una nube de palabras y diagramas Ven.
q) Tarjetas de reacciones faciales emocionales durante las entrevistas (las caras serán evaluadas en una escala Likert de 5 puntos desde totalmente en desacuerdo a totalmente de acuerdo).
r) Opiniones subjetivas acerca de T.

Se ha decidido utilizar el cuestionario SUS porque este sencillo cuestionario entrega los resultados más fiables [14]. Se ha complementado el cuestionario SUS con otras medidas debido a que los cuestionarios tienen limitaciones conocidas. En una revisión, Hornbaek [15] concluye que las medidas de satisfacción deben extenderse más allá de los cuestionarios.

Se han ampliado los cuestionarios con nuevos métodos para medir la satisfacción desarrollados por Microsoft, como por ejemplo las tarjetas reacción y cuestionarios de caras [16].

3.5. Protocolo - ¿Cómo ejecutar el estudio?

En la **figura 1** se presentan los pasos necesarios. Si se pueden inyectar fallos en el sistema, hay que tener cuidado de que [17]:

- Los fallos sembrados son similares a los fallos que se producen de forma natural en los programas reales debido a los errores cometidos por los desarrolladores.
- Deberían inyectarse bastantes fallos, es decir, un número suficiente de casos de cada tipo de fallo se ha sembrado.

El tipo y el procedimiento para ejecutar cada caso de estudio depende de las respuestas dadas a las preguntas de la **sección 3.4**. Esto se traduce en 7 posibles escenarios de casos de estudio (ver **sección 3.4** (S4) para recordar el significado de la T_{known}, $T_{unknown}$, TS_C, TS_N):

Escenario 1. Consiste en una evaluación cualitativa. Dado que no se sabe cuántos errores hay, no se puede comparar con otras técnicas, ni tampoco se puede realizar una línea base de la empresa, no se puede hacer una evaluación cuantitativa. Sin embargo, estudiar y reportar las medidas encontradas de eficacia, eficiencia y satisfacción subjetiva se llevará a cabo durante entrevistas semi-estructuradas con profesionales de pruebas.

Escenario 2. Consiste en Escenario 1 y análisis cuantitativo basado en la línea base de la empresa. La medida en que esto es posible y cuán válidas son las conclusiones dependerá de los datos que se presentan en la línea base de la empresa.

Escenario 3. Consiste en (Escenario 1 o Escenario 2) y análisis cuantitativo de la Tasa de Detección de Fallos (FDR) respecto al conjunto conocido de fallos.

Escenario 4. Consiste en (Escenario 1 o Escenario 2) y comparación cuantitativa de T y TS_C. Dado que TS_C ya existe, hay algunas medidas que no se pueden comparar (por ejemplo, en relación a la creación/diseño de la *suite* de pruebas, etc.), que serán cubiertas con el análisis del Escenario 1.

Escenario 5. Consiste en Escenario 4 y FDR de T y TS_C. Este escenario agrega una comparación cuantitativa de la tasa de detección de fallos de T con TS_C al Escenario 4.

Escenario 6. Consiste en (Escenario 1 o Escenario 2) y comparación cuantitativa de T y (T_{known} or $T_{unknown}$).

Escenario 7. Consiste de Escenario 6 y FDR de T y (T_{known} or $T_{unknown}$).

4. Casos de aplicación del *framework*

Esta sección describe tres instancias del *framework*. Las experiencias son diferentes pero aun así, la instanciación del *framework* resulta sencilla.

En los sub-apartados siguientes se muestra cómo hemos aplicado el *framework* de manera efectiva en todos estos ejemplos.

4.1. Pruebas estructurales basadas en búsquedas

En el trabajo publicado en [18] se presenta una instancia del *framework* sobre un caso de estudio. El objetivo de este trabajo era investigar la escalabilidad de las pruebas estructurales basadas en búsquedas desarrolladas en el proyecto EvoTest [6] y que se automatizaron con una herramienta llamada ETF (*Evolutionary Testing Framework*).

La herramientas fue evaluada con dos compañías (Daimler[1] y Berner&Mattner[2]) quienes participaron en el estudio por ser socios del proyecto EvoTest.

Las cuestiones planteadas en la investigación relacionadas con la efectividad, eficiencia y satisfacción del usuario fueron:

- En comparación con las pruebas aleatorias, EFT resulta más efectivo y eficiente encontrando casos de pruebas en sistemas reales.
- La cantidad de tiempo, esfuerzo y conocimiento necesario para configurar y usar ETF lo hace complejo en los entornos industriales.

Este caso de estudio se dirigió hacia los técnicos de pruebas empleados en las empresas participantes (sujetos). En el caso del objeto del estudio, fueron funciones C seleccionadas de sistemas reales.

Sin embargo, dado que las compañías no podían ofrecernos información sobre los bugs existentes ni podían incluir fallos, la respuesta a las preguntas S1 y S3 de la **sección 3.4** fue NO. La respuesta a la cuestión S4 fue sí pero sólo con otra técnica que pude ser utilizada con un mínimo esfuerzo humano, como es por ejemplo la técnica de pruebas aleatorias.

Las variables de la **sección 3.5** fueron medidas a través del número de casos de pruebas, el grado de código cubierto, el tiempo necesario para personalizar ETF, el tiempo para generar los casos de prueba, el tiempo para probar el sistema, y la satisfacción subjetiva y cualitativa obtenida en entrevistas informales con los equipos en las empresas.

> “Este caso de estudio se dirigió hacia los técnicos de pruebas empleados en las empresas participantes”

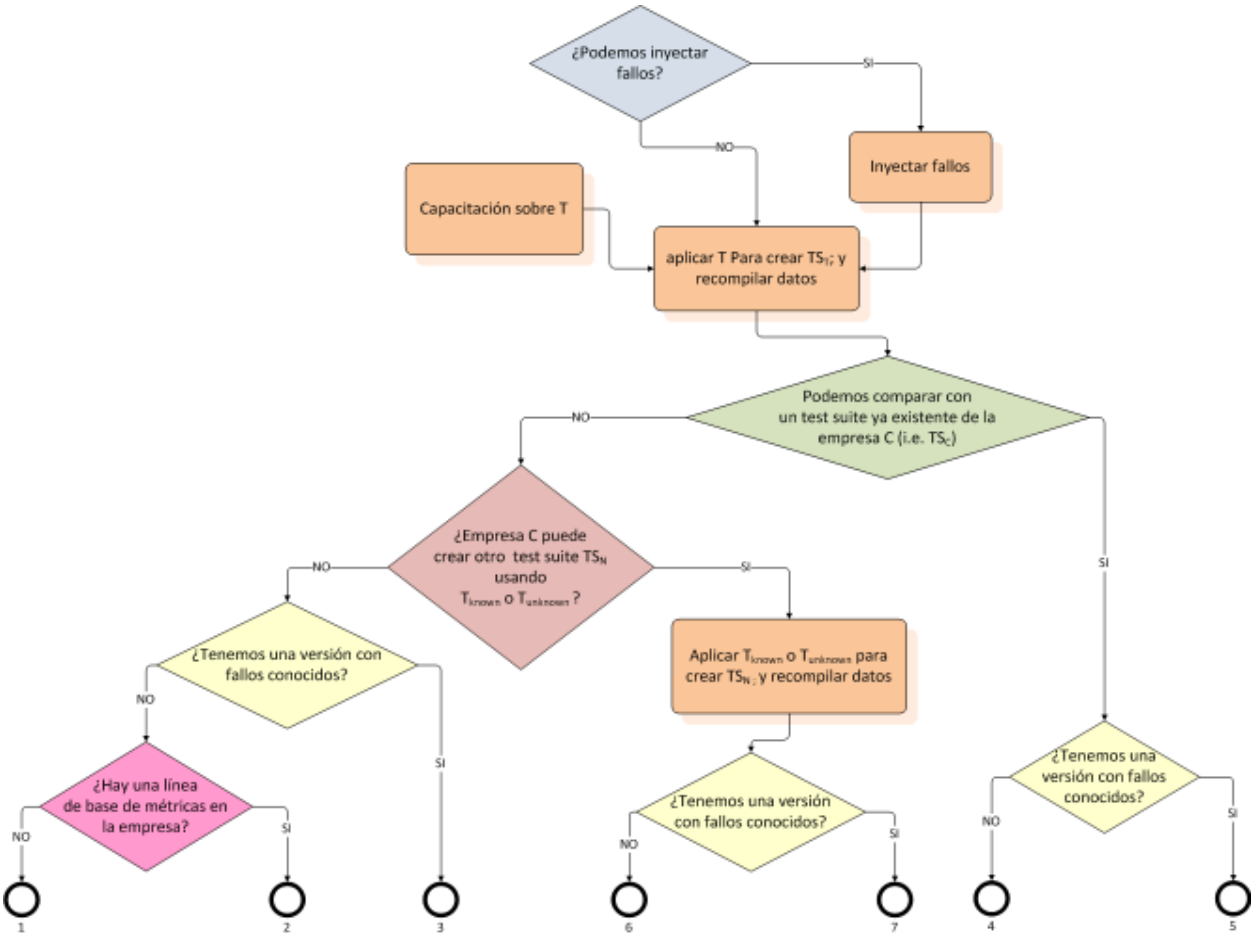

Figura 1. Posibles escenarios de casos de estudio.

Dado que no se podían prever los fallos a encontrar puesto que el software estaba bajo producción, se midieron los valores de progreso para conseguir una medida cualitativa de la calidad de los casos de prueba.

La instancia de ***do*** de la **sección 3.6** y **figura 1** consiste en: (1) Instalar y configurar ETF de acuerdo a su manual. Durante estas actividades, el trabajo diario medía y contenía una lista de tareas (que incluye aspectos como fecha, tiempo y descripción) que eran ejecutadas de acuerdo a ETF (por ejemplo, instalación, configuración, encontrar un conjunto apropiado de la evolución, etc.) (2) Ejecutar cada búsqueda 30 veces para asegurar las medidas conseguidas y asociar los datos a las variables seleccionadas. (3) Tener entrevistas informales sobre la adecuación y aceptabilidad en el entorno empresarial.

4.2. Pruebas funcionales basadas en búsquedas

En [19] se usó una instancia del *framework* para evaluar un caso de estudio cuyo principal objetivo era ver la aplicabilidad de pruebas funcionales basadas en las búsquedas.

Las pruebas funcionales no estaban completamente automatizadas (como en el caso de las pruebas estructurales de la sección anterior) y, de nuevo, las empresas indicaron que al ser un sistema en producción sería complejo encontrar fallos y que no se podrían generar más fallos.

De esta forma, las preguntas planteadas fueron:

Efectividad:

- ETF aplicado al mundo real, permite generar mejores casos de pruebas en comparación con otras técnicas como las pruebas aleatorias.
- ETF es más efectivo encontrando errores, cuando se aplica en entornos reales para pruebas de caja negra, que para pruebas aleatorias.

Satisfacción del usuario y eficiencia:

- Es posible utilizar ETF sin un conocimiento detallado del evolutivo para buscar las pruebas de interés.
- Después de la instalación de ETF, la cantidad de tiempo y esfuerzo que se necesita para configurarlo es abordable en un entorno industrial.

En este caso, el sujeto estaba compuesto por tres *testers* de sistemas empotrados en cada una de las dos empresas. Los *testers* ya tenían un conocimiento previo de los principios de pruebas evolutivas. Los objetos eran los sistemas de control integrados reales en el campo del automóvil.

Una vez más, las empresas no podrían compartir información acerca de los errores existentes, conjuntos de pruebas existentes, ni podían añadir fallos ni pruebas por encontrarse con sistemas en explotación, por lo que la respuesta a las preguntas de S1 a S4 de la **sección 3.4** fueron NO. Así, el escenario de la **sección 3.6** se corresponde en este caso con el número 1.

La calidad de los casos de pruebas se analizaba en función de los resultados conseguidos. Por último, indicar que se añadió una comparación cuantitativa de las pruebas al azar para tener una línea de base para la comparación de los algoritmos de búsqueda subyacentes.

Las variables que se midieron fueron el número de casos de prueba, el número de casos de pruebas no válidos, el número de fallos encontrados, el tiempo necesario para configurar ETF, el tiempo para generar los casos de prueba, el tiempo para testear el sistema y la evaluación subjetiva conseguidas de las entrevistas en el entorno industrial.

Como no se esperaba encontrar fallos en el sistema, usamos la medida de la conveniencia de las pruebas en función de la calidad de las mismas.

La instancia de ***do*** de la **sección 3.6** y **figura 1** consistió en: (1) Instalar y configurar de acuerdo al manual de usuario de ETF. El mantenimiento de las trabajos diarios incluía las tareas (incluyendo fecha, tiempo y descripción) que eran ejecutadas para configurar el manual de ETF (por ejemplo, tareas para encontrar un conjunto apropiado de parámetros, etc.); (2) Implementar los componentes de los casos de estudio (por ejemplo, especificaciones individuales y los drivers de prueba). (3) Definir, redefinir e implementar las funciones de completitud y validar su adecuación para la gestión de los requisitos y los trabajos diarios (4) Ejecutar las búsquedas 30 veces para dar un valor estadístico adecuado y recopilar los datos. (5) Tener entrevistas con los *testers* en las empresas para conocer su opinión.

4.3. Web testing de aplicaciones AJAX

En [20] el *framework* se instanció para evaluarlo en base a 4 técnicas de prueba que se ejecutan de manera frecuente en pruebas web: pruebas basadas en modelos, técnicas basadas en medidas de cobertura, técnicas de caja negra y pruebas basadas en estado.

Las técnicas fueron evaluadas por académicos y los objetos de evaluación fueron aplicaciones web obtenidas en un libro de estudio de aplicaciones industriales. Dado que el estudio fue realizado por académicos, no se pudieron hacer medidas subjetivas de satisfacción así que las preguntas de investigación fueron:

- ¿Que efectividad tienen la pruebas de técnicas de pruebas en Web para la detección de fallos?
- ¿Cuál es el esfuerzo requerido para aplicar cada una de las técnicas?

Teniendo en cuenta que no era software industrial, podíamos incluir fallos así que el escenario de la **sección 3.6** en este caso se corresponde con el número 3. Las variables que se midieron fueron: número de fallos encontrados, cobertura de los casos de uso, tipos y criticidad de los fallos que se encontraron, tamaño de los paquetes de prueba, tiempo (en horas/hombre) necesarias para testear la infraestructura específica así como la complejidad de los paquetes de prueba.

La instanciación de ***do*** de la **sección 3.6** consistió en: (1) Inyectar fallos (esta actividad era realizada por una persona diferente al que probaba) dentro de las aplicaciones web originales, intentando simular los errores de la programación usando la taxonomía o defectos de [20]. Estos cambios no producían errores graves en la aplicación pero fallaban en el comportamiento esperado de la misma; (2) Aplicar las técnicas de prueba Web seleccionada para encontrar estos fallos derivando paquetes de prueba para cada uno de ellos; (3) Aplicar cada paquete de pruebas en las aplicaciones.

5. Conclusiones

En este trabajo se ha presentado un *framework* metodológico para evaluar técnicas de pruebas de software. El objetivo esencial del *framework* es permitir al equipo de pruebas facilitar la definición de los casos de estudio instanciando dicho *framework*, con la seguridad de que dicho *framework* se basa en trabajos empíricos. Además, como todos los casos de estudio se ejecutan en base al mismo diseño, será más sencillo comparar todos los resultados obtenidos permitiendo a los investigadores disponer de un grupo de experiencias que permitan mejorar el campo de investigación de las pruebas de software.

En el trabajo se han presentado tres ejemplos de aplicación exitosa del *framework* que nos ha permitido evaluar la aplicabilidad y efectividad de la propuesta. Recientemente se ha utilizado el *framework* para 3 estudios más que están pendientes de publicación. Nuestros trabajos futuros van encaminados hasta esta dirección, intentando conseguir más ejemplos para la validación del *framework*, redefinirlo y comenzar a crear nuestro repositorio de experiencias que mejoren la investigación en las pruebas de software.

Agradecimientos

Este trabajo ha sido parcialmente financiado por los proyectos: FITTEST (ICT257574, UE, 2010-2013), CaSA-Calidad (TIN2010-12312-E, Ministerio de Ciencia e Innovación) y Fondecyt TESTMODE (11121395, 2012-2015).

Referencias

[1] B. Kitchenham, T. Dyba, M. Jorgensen. "Evidence-based software engineering". *Proc of ICSE. IEEE, 2004*, pp. 273–281.

[2] N. Juristo, A. Moreno, S. Vegas. "Reviewing 25 years of testing technique experiments". *ESE., vol. 9, no. 1-2*, pp. 7–44, 2004.

[3] P. Runeson, C. Andersson, T. Thelin, A. Andrews, T. Berling. "What do we know about defect detection methods?". *IEEE Softw., vol. 23, no. 3*, pp. 82–90, 2006.

[4] N. Fenton, S. Pfleeger, R. Glass. "Science and substance: a challenge to software engineers". *Software, IEEE, vol. 11, no. 4*, pp. 86 –95, julio 1994.

[5] T. Vos, B. Marín, MJ Escalona, A. Marchetto. A Methodological Framework for Evaluating Software Testing Techniques and Tools. *12th International Conference on Quality Software (QSIC 2012)*: pp. 230-239, 2012.

[6] T. Vos. "Evolutionary testing for complex systems". *ERCIM News, vol. 2009*, no. 78, 2009.

[7] T. Vos. "Continuous evolutionary automated testing for the future internet". *ERCIM, vol. 2010, no. 82*, pp. 50–51, 2010.

[8] S. Eldh, H. Hansson, S. Punnekkat, A. Pettersson, D. Sundmark. "A framework for comparing efficiency, effectiveness and applicability of software testing techniques". *TAICPart*, pp. 159–170, 2006.

[9] C. Lott, H. Rombach. "Repeatable software engineering experiments for comparing defect-detection techniques". *ESE, vol. 1*, pp. 241–277, 1996.

[10] V. Basili, R. Selby, D. Hutchens. "Experimentation in software engineering". *IEEE TSE, vol. 12*, pp. 733–743, 1986.

[11] H. Do, S. Elbaum, G. Rothermel. "Supporting controlled experimentation with testing techniques: An infrastructure and its potential impact". *ESE, vol. 10, no. 4*, pp. 405–435, 2005.

[12] B. Kitchenham, S. Linkman, D. Law. "Desmet: a methodology for evaluating software engineering methods and tools". *Computing Control Engineering Journal, vol. 8, no. 3*, pp. 120 –126, June 1997.

[13] J. Brook. "Sus: a 'quick and dirty' usability scale". En *Usability Evaluation in Industry*. Taylor and Francis, 1996.

[14] A. Bangor, P. Kortum, J. Miller. "An empirical evaluation of the system usability scale". *International Journal of Human-computer Interaction, vol. 24*, pp. 574–594, 2008.

[15] K. Hornbaek. "Current practice in measuring usability: Challenges to usability studies and research". *Int. J. Hum.-Comput. Stud., vol. 64*, pp. 79–102, febrero 2006.

[16] J. Benedek, T. Miner. "Measuring desirability: New methods for measuring desirability". *Proc of the Usability Professionals Association Conf*, 2002.

[17] A. Memon, Q. Xie. "Studying the fault-detection effectiveness of gui test cases for rapidly evolving software". *IEEE TSE , vol. 31, no. 10*, pp. 884 – 896, Oct. 2005.

[18] T. Vos, A. Baars, F. Lindlar, P. Kruse, A. Windisch, J. Wegener. "Industrial scaled automated structural testing with the evolutionary testing tool". *ICST, 2010*, pp. 175–184.

[19] T. Vos, F. Lindlar, B. Wilmes, A. Windisch, A. Baars, P. Kruse, H. Gross, J. Wegener. "Evolutionary functional black-box testing in an industrial setting," *Software Quality Journal*, pp. 1–30, 2012.

[20] A. Marchetto, F. Ricca, P. Tonella. "A case study based comparison of web testing techniques applied to ajax web applications". *International Journal on Software Tools for Technology Transfer, vol. 10*, pp. 477–492, 2008.

Notas

[1] <http://www.daimler.com>.

[2] <http://www.berner-mattner.com>.

Celestina Bianco
Systelab Technology S.A.
<celestina.bianco@systelabsw.com>

Medición de pruebas para la mejora de la calidad y la eficiencia

Resumen: *Las métricas útiles para soportar el proceso de toma de decisiones son generalmente simples, siendo más potentes cuanto más están basadas en datos validados, posiblemente recopilados dentro de la organización. Permiten, por ejemplo, decidir el compromiso que cumple los objetivos de calidad, tiempo de puesta en mercado y costes de mantenimiento, basados en las dimensiones de un producto, el tiempo promedio para corregir defectos y los resultados iniciales de las pruebas de una aplicación. Este artículo trata esta importante área, mediante un recorrido a través de un caso de estudio real de estimación para el software embebido en un dispositivo médico.*

Palabras clave: *Calidad y test, caso de estudio, coste/beneficio, métricas y estimaciones, proceso.*

Autora

Celestina Bianco es directora de calidad en Systelab Technology S.A., una compañía privada que proporciona servicios de ingeniería y calidad adaptada a necesidades del cliente en mercados de productos sanitarios y salud, donde el proyecto, proceso y gestión de la calidad son críticos. Es titulada en Física y con 4 años de experiencia de trabajo en inteligencia artificial. Desde 1987 ha trabajado en el área de software para la salud, primero en I+D y posteriormente en el área de calidad, dando soporte a la definición y control de los sistemas calidad tanto en Systelab como en otras organizaciones. En Systelab, gestiona el departamento de aseguración y control de la calidad. Participa activamente en conferencias internacionales, seminarios y participa en grupos de trabajo en organismos y estándares internacionales como FDA (*Food and Drug Administration*) e ISO (*International Organization for Standardization*).

1. Introducción

Un constructor de instrumentos de análisis clínico subcontrata el desarrollo de un nuevo instrumento. El proveedor desarrollará el hardware y el software siguiendo un ciclo de vida iterativo clásico. Para conseguir una validación independiente, las pruebas del software serán subcontratadas a una tercera compañía.

El cliente de este software definirá los requisitos en dos fases principales, y por lo tanto, el desarrollo. Planea realizar la validación al final de la segunda fase de desarrollo.

La compañía contratada para las pruebas del software sugiere una estrategia diferente, introduciendo pruebas en paralelo al desarrollo.

Los datos de literatura, de la experiencia de la compañía que realizará la validación, y resultados de una validación inicial informal convencen al cliente de las ventajas del enfoque sugerido. Los datos disponibles permiten una predicción de los esfuerzos necesarios en la primera fase y la calidad obtenida. Los datos coleccionados al final de la primera fase son usados para confirmar las estimaciones y refinarlas para la segunda fase. Al final de la segunda fase de desarrollo, el cliente dispone de datos relativos a la calidad actual del producto, estimación de costes y tiempo de lanzamiento al mercado respetando la calidad objetivo, así como una estimación de costes de mantenimiento correctivo necesarios para corregir los problemas que permanezcan abiertos en el momento de liberación del producto.

Al final del proyecto, los datos permiten al cliente calcular las ventajas reales (ahorro de tiempo y coste) del enfoque propuesto por el proveedor, y realizar decisiones informadas sobre futuros proyectos.

2. El proyecto

Un fabricante y distribuidor de componentes y fungibles para dispositivos médicos (el cliente) decidió extender su negocio hacia la producción y distribución de analizadores de sangre.

El modo en el que afectan a la salud pública, la calidad y rendimiento de estos instrumentos es dictada por el mercado y por normativas internacionales. El control final de un instrumento y la mitigación de los principales riesgos son generalmente proporcionados por el software, lo que causa que sea éste uno de los componentes más críticos.

Con una amplia y profunda experiencia clínica y química, el cliente definió los requisitos del sistema y la interfaz de usuario, subcontratando el desarrollo del hardware y componentes software a una compañía que tiene una amplia experiencia con hardware y software para sistemas embebidos.

Para cumplir con las normas de la FDA (*Food and Drug Administration*) que se aplican a dispositivos médicos, y para asegurar una adecuada calidad del software, el cliente subcontrató la validación de software a un tercero. Los equipos de proyecto de cada una de las tres compañías debían trabajar con una estricta colaboración entre ellos en todas las tareas de las que eran responsables en las diferentes fases del proyecto.

2.1. Contexto y precondiciones

El proyecto se divide en dos fases principales. Al final de la primera el instrumento dispondrá de las características mínimas que le permitirán ser utilizado para evaluar las funcionalidades químicas y analíticas por técnicos de laboratorio. Las características adicionales que necesitan los usuarios serán desarrolladas en una segunda fase.

El cliente definirá los requisitos del sistema y la parte principal de los requisitos de interfaz de usuario antes de comenzar el desarrollo del software. Durante la fase 1 el cliente definirá los requisitos del sistema que serán implementados en la segunda fase.

En cada fase, la primera tarea de la compañía encargada del desarrollo de software será la definición progresiva de las especificaciones software de la fase y su revisión con el cliente.

La compañía que realizará la validación participará en todas las fases y revisiones para realizar el análisis de riesgos.

2.2. Periodo de prueba para el enfoque

El cliente, de acuerdo con los desarrolladores, solicitó que la validación se realizase hacia el final del proyecto. La compañía a cargo de la validación sugirió un enfoque alternativo: una verificación cíclica en paralelo al desarrollo.

Después de tres meses de pruebas funcionales del software en paralelo al desarrollo por parte de los desarrolladores, el número de defectos encontrados fue 256, habiendo probado aproximadamente 280 requisitos software.

2.3. Estimación del coste de la falta de calidad

Es conocido de la literatura, y conformado

> “El cliente, de acuerdo con los desarrolladores, solicitó que la validación se realizase hacia el final del proyecto. La compañía a cargo de la validación sugirió un enfoque alternativo: una verificación cíclica en paralelo al desarrollo”

por la propia experiencia de la compañía que realiza la validación, que el coste de eliminación de un defecto detectado durante el desarrollo es en promedio de 2 horas, mientras que si es detectado tras la integración final es alrededor de 8 horas.

Con las pruebas iniciales, el tiempo ahorrado se estimó por lo tanto en unos 3 meses del equipo, ¡alrededor del mismo tiempo empleado en desarrollo!

En este punto, se realizó una estimación aproximada del ratio defectos/tamaño en 0,9 defectos/requisito. En la fase 1 se definieron alrededor de 1.000 requisitos; es decir, se esperaba que las pruebas de la fase 1 detectasen aproximadamente 900 defectos.
El coste de no conformidad debido a los defectos no detectados durante la fase 1 se estimó en más de 5.000 persona-hora, es decir, más de nueve meses de trabajo de las tres personas del equipo dedicadas a desarrollo y prueba del producto.

Los siguientes gráficos ilustran las estimaciones realizadas basándose en los datos disponibles: La **figura 1** muestra los defectos esperados y la **figura 2** el ahorro estimado en el tiempo de desarrollo total del proyecto al poner los esfuerzos de pruebas desde el principio de los primeros pasos de la fase 1.

Considerando que la mayor parte de los defectos serán corregidos antes de la liberación del dispositivo médico, estos defectos detectados al final del desarrollo supondrían un mes de retraso, con el correspondiente sobrecoste en recursos. Este retraso puede ser evitado organizando y diseñando las pruebas funcionales en paralelo al desarrollo.
La resistencia a este enfoque surgió de la siguiente consideración: las especificaciones no son estables y habrá cambios solicitados que “invalidarán” las pruebas realizadas.

La contestación a esta duda es doble:
- Si las partes desarrolladas son probadas tan pronto están disponibles, los cambios serán realizados sobre una base conocida y estabilizada; la prueba de los cambios estará enfocada a éstos y será más fácil.
- El impacto de un cambio puede ser evaluado antes de ser aprobado.

Estas dos consideraciones convencieron de que la prueba en paralelo tenía la ventaja por un lado de hacer la aplicación y la especificación más estables y controladas, y por el otro de asegurar que el estado de ambas es conocido en cualquier momento. Esto venció la resistencia inicial y la prueba en paralelo fue iniciada formalmente.

3. Primera fase

3.1. Recopilación, validación y elaboración de datos

Al final de la fase 1 el total de requisitos definidos fue 1.028. De acuerdo con la estimación (0,9 defectos/requisito) los defectos estimados serían 925.
Los defectos fueron gestionados con el soporte de una herramienta de gestión de defectos. Al final de la fase 1 se registraron 1.100. La realidad fue peor que la estimación, los defectos detectados fueron alrededor de 16% más que los esperados, por lo que el ratio ajustado defecto/requisito sería alrededor de 1,1 en lugar de 0,9.

¿Qué fue incorrecto en la estimación? ¿Que significa esto en términos de costes y ahorros?

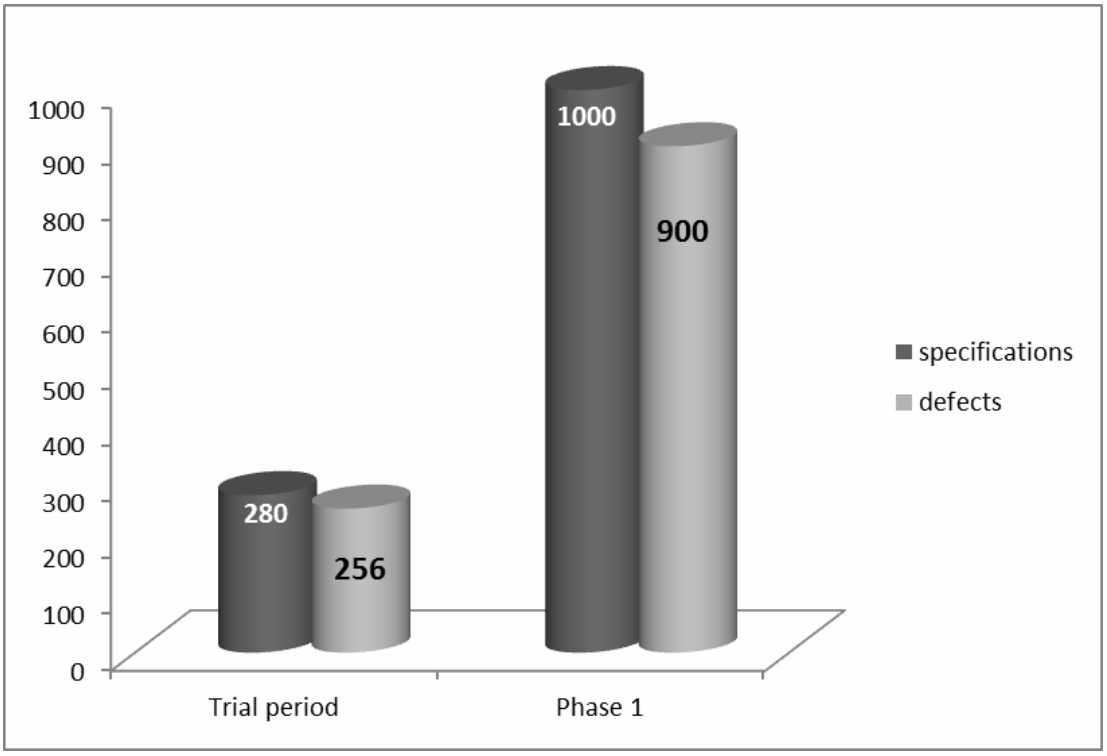

Figura 1. Defectos detectados y esperados.

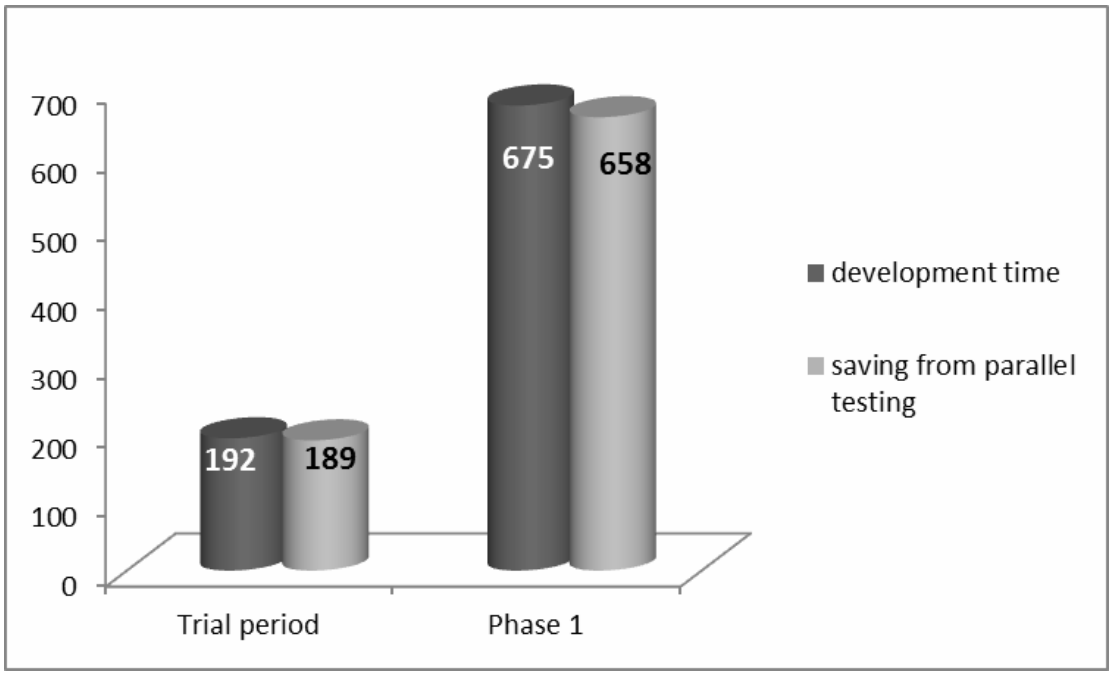

Figura 2. Esfuerzos globales en la fase 1 (personas-mes).

3.2. Análisis de los resultados

El análisis de los datos reveló que las especificaciones habían sufrido muchos cambios y, por lo tanto, los requisitos realmente implementados y probados eran más de los contabilizados al final de la fase 1.

Desafortunadamente, no todos los cambios fueron seguidos formalmente, y no fue posible calcular el número exacto de requisitos implementados. Sin embargo, estimando el número total de cambios respecto de los que han tenido el seguimiento, el ratio defecto/requisito es más cercano a la estimación inicial, lo que se representa en la **figura 3**.

El resultado anterior es relevante, puesto que muestra que **la calidad del producto no es afectada por el número de cambios si la aplicación es robusta**, debido al hecho de que los cambios son realizados sobre una base conocida y el impacto de éstos puede ser estimada con adelanto.

El análisis detallado de los resultados reveló también que la tendencia de la severidad de los defectos detectados no fue homogénea, ni tampoco la tendencia de las correcciones. Es decir, durante las primeras semanas de pruebas los defectos detectados fueron más severos que los detectados posteriormente, y el porcentaje de defectos corregidos sobre los defectos detectados fue menor en el periodo inicial. Este análisis hizo evidente que los defectos de alta severidad impiden la detección de otros problemas, y que **cuanto más rápida es la solución de los defectos más severos, la prueba es más efectiva, y el producto más robusto**.

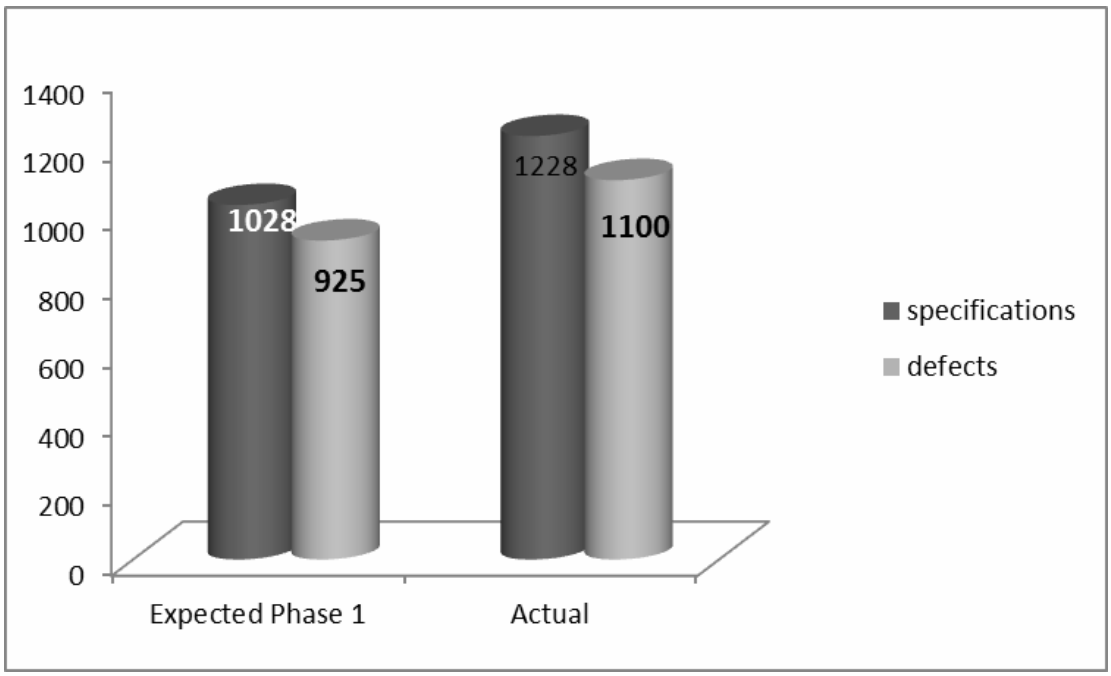

Figura 3. Comparación de defectos detectados con defectos reales en la fase 1, considerando cambios.

> “Cuanto más rápida es la solución de los defectos más severos, la prueba es más efectiva, y el producto más robusto”

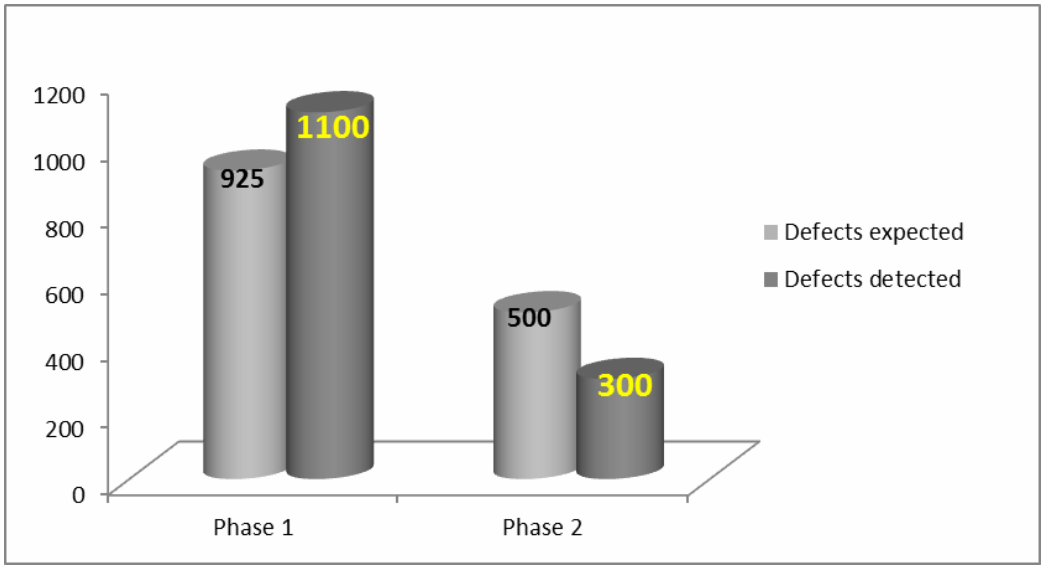

Figura 4. Comparación de defectos detectados con defectos reales en las fases 1 y 2.

4. Segunda fase

4.1. Estimación

En el momento de comenzar el desarrollo de la fase 1 los requisitos a implementar fueron 482. El ratio de cambios fue estimado aproximadamente igual al de la fase 1. La estimación de defectos esperados fue por lo tanto de alrededor de 500, aplicando el ratio ajustado 1:1,1.

4.2. Ajuste de la estrategia

Basado en el análisis de los resultados de la fase 1, la gestión de defectos fue modificada, dando mayor prioridad a corregir los defectos abiertos lo antes posible.

Otro cambio en la cadena de desarrollo y validación fue introducido como consecuencia de requisitos normativos. El software para dispositivos médicos debe ser aprobado por la FDA en el caso de que sea comercializado en Estados Unidos, y en Europa necesita una aprobación de la Comunidad Europea. Ambas aprobaciones se otorgan en base a la documentación del proceso aplicado durante el desarrollo del producto y a las evidencias de la validación. Las pruebas unitarias de componentes críticos son requeridas como parte de la evidencia de que se ha realizado una adecuada validación.

El diseño y ejecución de pruebas unitarias se hizo obligatorio comenzando en la fase 2. Los defectos de las pruebas unitarias no se registraron como los defectos de las pruebas funcionales.

4.3. Recopilación, validación y elaboración de datos

Al final de la ejecución completa de todas las pruebas de la fase 1, los defectos registrados fueron 300, alrededor del 40% menos de los esperados. Todas estas cifras se muestran en la **figura 4**.

La estimación fue de nuevo incorrecta y la desviación requirió un análisis.

4.4. Análisis de los resultados

El análisis de los datos realizado para entender la tendencia reveló que:

- La distribución en severidad de los defectos detectados se había hecho más homogénea que en la fase 1 a lo largo de las diferentes versiones internas probadas.
- El número de defectos de regresión (defectos abiertos como consecuencia de un cambio o corrección de otros defectos) fue más bajo.

El primer resultado es la consecuencia del cambio en el enfoque de la gestión de defectos. **El ratio de defectos de regresión ha descendido del 20% al 10% al introducir pruebas unitarias**.

El segundo resultado es probablemente consecuencia de las pruebas unitarias, puesto que los defectos detectados por pruebas unitarias se solucionaron antes de llegar al test funcional.

Si utilizamos los ratios de coste de resolución de defectos, llegamos a la conclusión de que se consiguió un ahorro adicional. Consideremos que la mitad del incremento de la estimación es debida a las pruebas unitarias. Si usamos el ratio 1:4 del coste de resolución de defectos detectados en pruebas unitarias respecto de los detectados en pruebas funcionales obtenemos una mejora adicional en el tiempo del proyecto de:

CostePruebasFuncionales – CostePruebas Unitarias = 100 * (2 - 0,5) = 150 horas

Extendiendo las métricas a todo el ciclo de vida, si las pruebas unitarias pueden detectar alrededor de un tercio de los defectos, sobre 1.400 defectos encontrados, aplicando las pruebas unitarias desde el principio, el ahorro podría ser 1.400 * 1,5 = 2.100 horas, que son 4 meses de trabajo para los tres miembros del equipo.

4.5. Planes para el mantenimiento

Al final de la validación de la fase 2 los defectos todavía no corregidos son 70. Esto significa que los defectos que necesitan ser corregidos para liberar un software con 0 defectos serían 77.

El coste de corrección de un defecto es de aproximadamente 2 horas en este punto. Sería al menos 4 veces superior si quedase abierto en el momento de la liberación, considerando costes adicionales de documentación, distribución, etc. Estas estimaciones se muestran en la **figura 5**.

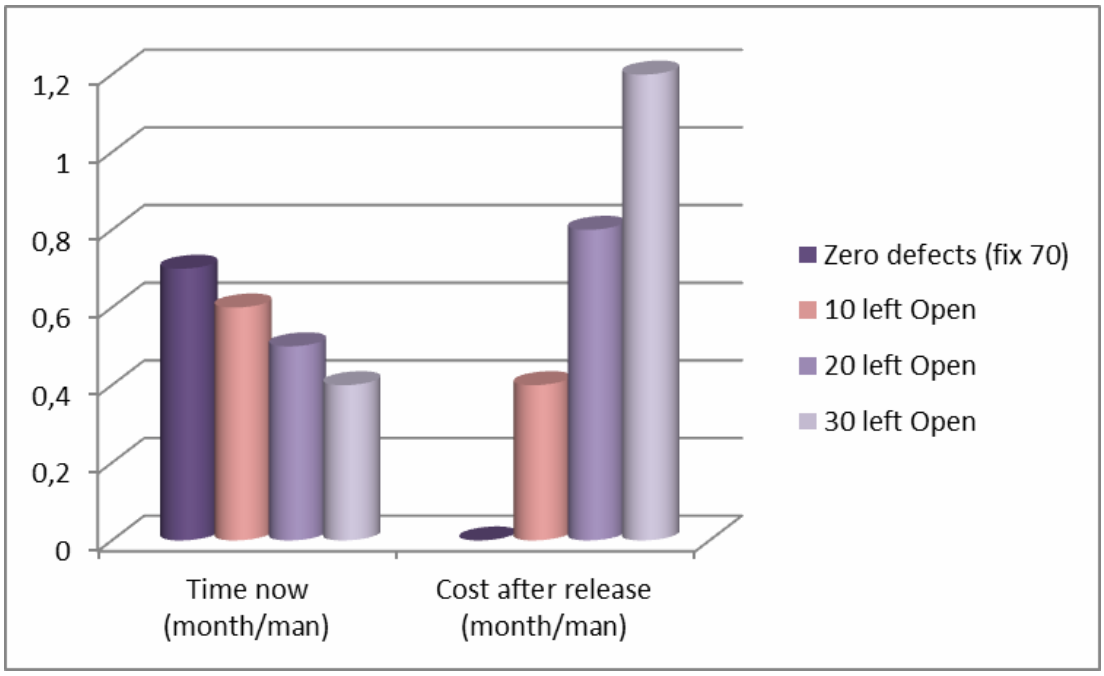

Figura 5. Comparación del retraso hasta la liberación (*time now*) y coste del mantenimiento correctivo dependiendo del número de defectos pendientes de corrección (*cost after release*).

La decisión del fabricante es liberar el producto con 20 defectos abiertos de bajo riesgo, lo que significa que tiene que planificar el 50% de una persona-mes para corregir algunos defectos antes de la primera liberación al mercado, y preparar el equipo para la solución en breve plazo de los restantes.

Por medio del conocimiento del coste de desarrollo, prueba y corrección para los requisitos, el fabricante podrá fácilmente evaluar el coste de cualquier cambio solicitado, así como el instante apropiado de lanzamiento al mercado.

5. Conclusiones

Cuando la compañía encargada de validar la aplicación sugirió comenzar la verificación en paralelo con el desarrollo, a pesar de disponer de una especificación de requisitos incompleta con muchos cambios esperados, el propósito fue doble: por una parte suministrar mayor visibilidad del estado de la aplicación para dar soporte a decisiones informadas; por otra parte, minimizar los esfuerzos de detección y corrección de defectos.

Las métricas recopiladas y elaboradas confirmaron la hipótesis inicial y las tendencias esperadas. Adicionalmente, el análisis de los datos sugirió un cambio en el proceso: la introducción de pruebas unitarias antes de la integración, y una evaluación de los cambios más cuidadosa basada en datos.

La recolección y análisis de datos hicieron más fácil la monitorización del proyecto y la planificación de las liberaciones conociendo el compromiso de coste en términos de calidad, riesgos, retraso en la puesta en mercado y costes tras la liberación.

5.1. Conclusiones en el cliente

El cliente comprendió el valor de tener en cualquier momento el estado de la aplicación bajo control. Se apreció evidente que con un proceso controlado es fácil obtener métricas que soporten las decisiones estratégicas, es posible estar más seguro de la estabilidad de esa "desconocida caja negra" que se considera generalmente el software, y mantener los cambios implementados bajo control. Además, se facilita la planificación de pruebas de campo, la formación y la liberación al mercado.

El fabricante puede también planificar con mayor precisión el mantenimiento técnico: corrección de defectos, cambios, mejoras y extensiones en términos de recursos, costes y tiempo.

Por otra parte, disponiendo de un claro conocimiento de la calidad, fue posible planificar de forma precisa la dotación de personal del equipo de servicio.

5.2. Conclusiones en la compañía responsable de la validación

La compañía encargada de la validación reconfirmó su capacidad para suministrar servicios con un alto grado de definición y estimaciones precisas.

Las sugerencias informadas que realizaron y la confirmación de las hipótesis y estimaciones abrieron una colaboración con el cliente para otros proyectos, y la posibilidad de sugerir mejoras adicionales al proceso.

5.3. Los pasos siguientes

Como consecuencia de la confianza conseguida en la compañía proveedora de la validación del software, el cliente le asignó el desarrollo y mantenimiento de las siguientes versiones de software.

Del enfoque aplicado en las fases 1 y 2 se apreció como natural el paso a la adopción de metodologías ágiles. Los datos recolectados y el análisis constituyeron una entrada natural para la definición de "listo" y "hecho".

Los buenos resultados obtenidos y cuantificados con la adopción de pruebas unitarias y la corrección de defectos tan pronto son detectados, encaminan de forma natural a paradigmas de desarrollo dirigido por pruebas.

El enfoque ágil "puro" fue modificado en dos aspectos: a) para requisitos normativos, el volumen de documentación producido es alto; b) Dada la experiencia aquí descrita, se ejecuta una prueba de regresión funcional completa en cada sprint para mantener el sistema estable.