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"

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”

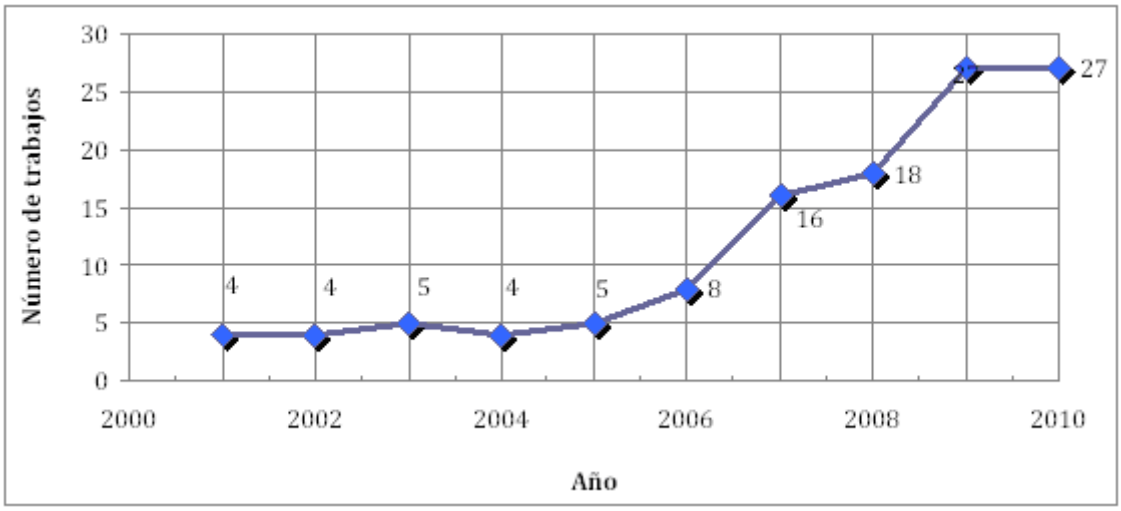

Figura 1. Número de trabajos de priorización por año.

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-

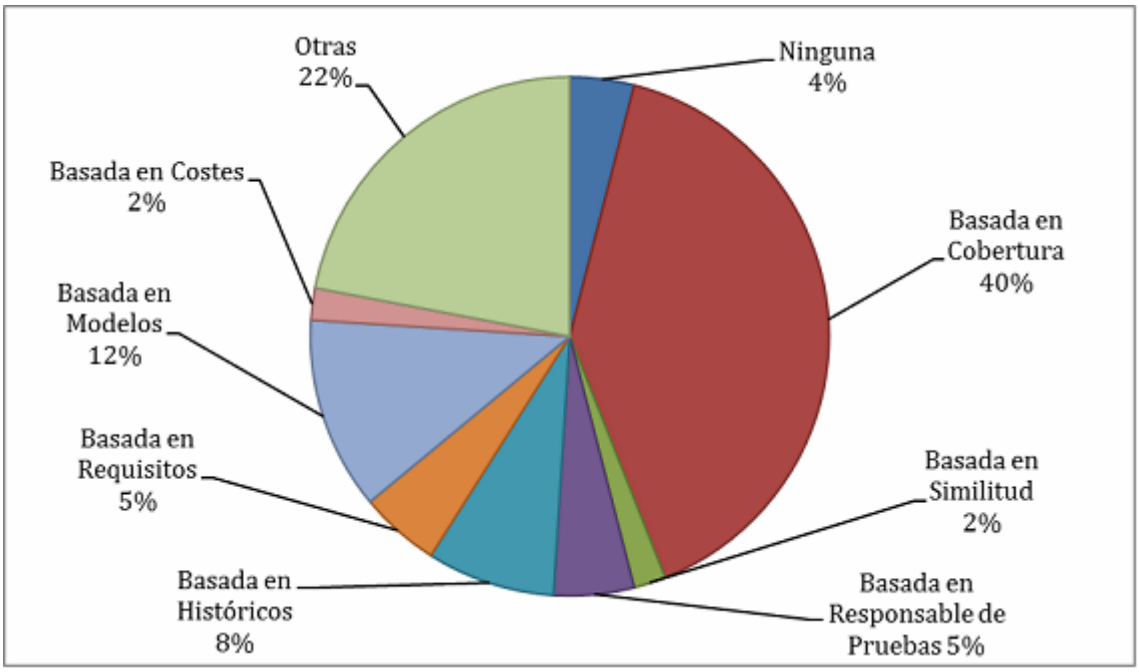

Figura 2. Distribución de propuestas de priorización.

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>.