Tesis Doctoral

Criterios basados en abstracciones de comportamiento para testing de conformidad de protocolos

Czemerinski, Hernán

2015-04-14

Universidad de Buenos Aires
Facultad de Ciencias Exactas y Naturales
Departamento de Computación

Criterios basados en abstracciones de comportamiento para testing de conformidad de protocolos

Tesis presentada para optar al título de Doctor de la Universidad de Buenos Aires en el Área Ciencias de la Computación

Hernán Czemerinski

Director de tesis:	Dr. Sebastián Uchitel
Consejero de estudios:	Dr. Sebastián Uchitel

Fecha de defensa: 14 de abril de 2015

Buenos Aires, 2015

Criterios basados en abstracciones de comportamiento para testing de conformidad de protocolos

Resumen:

Los artefactos de código que tienen requerimientos no triviales con respecto al orden en el que sus métodos o procedimientos deben ser invocados son comunes y aparecen, por ejemplo, como implementaciones de APIs y objetos. El testeo de que dichos artefactos se ajusten a su *protocolo esperado* es un problema importante y desafiante. En esta tesis se proponen y estudian criterios de adecuación de testing de conformidad basados en cubrir una abstracción de la semántica del comportamiento esperado. Por lo tanto, los criterios son independientes tanto del lenguaje de especificación y las estructuras sintácticas usadas para describir el protocolo esperado como del lenguaje utilizado para implementarlo. En consecuencia, los resultados pueden ser de utilidad para diversos enfoques de caja negra para el testeo de conformidad de protocolos. Los resultados experimentales muestran que los criterios propuestos son buenos predictores de detección de fallas de conformidad y de criterios de cobertura estructurales clásicos como cobertura de sentencias y ramas. Además, también muestran que la división del dominio derivado de los criterios propuestos produce subdominios densos en fallas, y que al priorizar casos de test de acuerdo con los criterios propuestos se tiende a producir ordenamientos que generan una detección temprana de fallas de conformidad de protocolos.

palabras clave: testing de software; conformidad de protocolos; criterios de adecuación; partición de dominios; priorización de casos de test.

Behaviour Abstraction Based Adequacy Criteria for Protocol Conformance Testing

Abstract:

Code artefacts that have non-trivial requirements with respect to the ordering in which their methods or procedures ought to be called are common and appear, for instance, in the form of API implementations and objects. Testing such code artefacts to gain confidence in that they conform to their intended *protocols* is an important and challenging problem. In this thesis we propose and study experimentally conformance testing adequacy criteria based on covering an abstraction of the intended behavior's semantics. Thus, the criteria are independent of the specification language and structure used to describe the intended protocol and the language used to implement it. As a consequence the results may be of use to black box conformance testing approaches in general. Experimental results show that the criteria are a good predictor for fault detection for protocol conformance and for classical structural coverage criteria such as statement and branch coverage. Additionally, they also show that the division of the domain derived from the criterion produces subdomains such that most of its inputs are fault-revealing, and that prioritising test cases according to the abstraction coverage achieved tends to produce orderings that lead to earlier detection of protocol conformance failures.

keywords: Software Testing; Protocol Conformance; Adequacy Criteria; Partition Testing; Test Case Prioritisation.

Agradecimientos

Al escribir estas palabras tengo la impresión de estar terminando un muy largo viaje. No suele gustarme la palabra balance, pero lo cierto es que finalizar esta tesis representa mucho más que finalizar esta tesis. Engloba, por sobre todo, muchos años, mucha gente y muchas experiencias. Fueron años muy buenos, divertidos y de emociones fuertes.

Quiero comenzar agradeciendo a mis directores Sebastián Uchitel y Víctor Braberman, ya que haber llegado a este momento se los debo en primer lugar a ellos. No creo estar diciendo nada novedoso al afirmar que son dos científicos de primer nivel. En ese sentido es un enorme privilegio que hayan sido ellos quienes guiaron el trabajo realizado durante estos años. Expertos en el área, marcaron las líneas para llevar adelante todo el proceso con mucha cabeza, mucha paciencia y gran intuición. Pero más allá de su calidad profesional, en este momento quiero hacer hincapié en que se trata de dos personas de una calidad humana extraordinaria. Hemos compartido muchas cosas y tengo por ellos un gran afecto. Sebas, Vic, aprendí un montón de ustedes. Muchísimas gracias.

Siguiendo un esquema *top-down* en lo que al trabajo refiere, quiero agradecer muy especialmente a todos mis compañeros de LaFHIS. Empiezo por Guido de Caso por haber colaborado directamente con los resultados presentados en esta tesis. Crack total. A Nicolás D'ippolito, porque si no hubiera sido por un cruce fortuito entre nosotros ésto ni siquiera habría arrancado. A Fernando Asteasuain, Esteban *concorde* Pavese y Gervasio Pérez, tres grandes bastiones de estos años, todos ellos miembros de la vieja guardia *lafhisiana*. Una mención muy especial para Daniela Bonomo, que siempre nos simplificó la vida a todos en todo. Genia. También le agradezco al resto del grupo: Germán Sibay, Diego Garbervetsky, Rodrigo Castaño, Mariano Cerrutti, Edgardo Zoppi, Guido Chari, Natalia Rodriguez, Ezequiel Castellano, Fernán Martinelli, Verónica Rodriguez, Daniel Ciolek, Fernando Schapachnik, Sven Stork y Sergio Yovine. Siempre fue un gusto ir a la facultad sabiendo que los iba a encontrar. Hemos pasado juntos años muy divertidos.

No quiero dejar de mencionar al Departamento de computación de la FCEyN. No creo equivocarme si digo que todos los que habitamos este espacio tenemos un lindo sentimiento por el lugar. Pero el departamento no es un ente abstracto, es el producto de la gente que lo compone. En lo personal me produce mucha satisfacción formar parte.

También agradezco a los jurados Hernán Melgratti, Andrés Diaz Pace y Juan Pablo Galeotti por haber aceptado evaluar el trabajo. Con una lectura profunda y muy lúcidas observaciones han contribuido a mejorar la versión final de esta tesis.

Agradezco a mis amigos de toda la vida: Adrián Kaminker, Martín Pines, Nicolás Avruj, Manuel Dubinsky, Mauro do Porto, Lucas Colonna, Andrés Farhi y Sebastián Schindel. El grupo que hemos formado es un capital enorme del que todos nosotros somos poseedores. También a Sergio Mera y Agustín Gravano, los dos grandes amigos que me dio la universidad y junto a quienes (en dos etapas distintas) compartí estos años en la facultad. A Irene Spivakow, con a quién a lo largo de los años que ha durado el doctorado hemos hecho otro gran trabajo. Agradezco a mi familia (ramas Czemerinski, Berestovoy, Mendiberri y Rojas) por estar siempre presente haciendo el aguante.

Por último, quiero agradecerles a Paula y a Manuel. Son el tesoro más preciado. Hemos formado una familia a la que amo profundamente. Pau, Manu, muchas gracias. A ustedes dedico esta tesis. Los amo.

Contents

Resumen en castellano

A continuación se encuentra un resumen de la presente tesis en castellano, presentando las ideas centrales de cada capítulo.

Capítulo 1: Introducción

Criterios de adecuación de protocols de APIs

A pesar de los progresos realizados durante los últimos años, la generación automática de casos de test eficientes y de alta calidad sigue siendo un desafío importante para muchos tipos de software [FA12, XXTdH11, HBB+09]. Este es el caso de componentes con estado tales como APIs, interfaces gráficas o clientes y servidores web que poseen requerimientos no triviales con respecto al orden en el que sus métodos o procedimientos debe ser invocados para producir resultados significativos o para acceder a determinadas funcionalidades [BKA11]. Un ejemplo de este tipo de componentes es la interfaz `ResultSet` de Java que se utiliza para acceder o modificar los elementos de una base de datos. El protocolo de invocación sobre una instancia de `ResultSet` prohíbe movimientos del cursor mientras se inserta un nuevo registro (es decir, no permite movimientos del cursor entre las llamadas a `moveToInsertRow` y `insertRow`), pero sí permite cerrar el `ResultSet` aunque la inserción no se haya completado. Otro ejemplo de componente con estado son algunos servidores web: a pesar de que deben ser capaces de recibir solicitudes en cualquier orden, ciertas funcionalidades son únicamente alcanzadas si se sigue una secuencia específica de acciones. En general, el testeo de componentes con protocolos de llamadas no triviales es particularmente difícil [XXTdH11, GZE+12]. Se sabe que son difíciles tanto para las enfoques random [GZE+12] como para técnicas de ejecución dinámica-simbólica [XXTdH11].

Un tipo de testing particularmente importante para los componentes con estado tiene como objetivo determinar si su comportamiento real se ajusta a su *comportamiento de invocación esperado* (ver por ejemplo [GKSB11]). La correcta implementación de un protocolo es crucial para obtener garantías de que código cliente no va a fallar debido a la realización de invocaciones a componentes que no implementan correctamente su protocolo esperado. En este contexto el testing se centra en la verificación de que el código bajo prueba acepta o rechaza secuencias de invocaciones a métodos de acuerdo con el protocolo de invocación esperado. Realizar este chequeo en forma completa no es posible, ya que en general implicaría hacerlo sobre un conjunto potencialmente infinito de secuencias de invocaciones.

Al utilizar criterios de adecuación de tests de caja blanca (es decir, criterios basados en el cubrimiento del código fuente) puede darse el caso que conjuntos de tests desarrollados durante el diseño de una aplicación resulten adecuados para ciertas implementaciones y no para otras. La pregunta que se pretende contestar en esta tesis es si es posible definir criterios de adecuación de testing de conformidad de protocolos que sean independientes del código a ser finalmente testeado. Es decir, si los criterios de adecuación pueden ser definidos en términos de su protocolo esperado en lugar de en una implementación.

El testing de caja negra ha abordado este problema hasta cierto punto. La gran mayoría de trabajos en testing de caja negra ha estudiado criterios de cobertura estructural sobre especificaciones [HBB+09]. Estos criterios son entonces definidos o bien en términos de elementos estructurales de especificaciones (por ejemplo [KKT07, OLAA03, Hie97, DF93, PRB09], etc.) o sobre código ejecutable generado a partir de especificaciones (por ejemplo [PPW+05, RCW08]). En consecuencia, en general se estudian criterios de cobertura que pueden estar influenciados por elementos accidentales de las especificaciones tales como predicados, elementos del flujo de control o elementos del flujo de datos. Diversos autores ya han advertido que aspectos accidentales de las especificaciones pueden tener influencia sobre la efectividad de los criterios (por ejemplo [PPW+05, RCW08, HGW04, RHOH98, MBLDP11, Wei10], etc.). Más aun, al estar las especificaciones estrechamente acopladas a un determinado lenguaje, la relación entre un criterio y el espacio de estados de un protocolo cubierto no suele estudiarse. Esto a su vez dificulta la generalización de los resultados empíricos existentes [HBB+09] al problema general de testing de caja negro de conformidad de protocolos.

Criterios de adecuación semánticos

A pesar de que una medida de cobertura semántica sería esperable como una medida natural de testing, existe una dificultad hasta ahora no explorada cuándo el comportamiento del sistema testeado posee un espacio de estados infinito. Uno de los principales desafíos al definir criterios de cobertura semánticos (o basados en comportamiento) es que el espacio de estados de protocolos de invocaciones es típicamente infinito y, en consecuencia, los criterios tradicionales de caja negra para conformidad de protocolos no pueden aplicarse porque asumen espacios de estados finitos [KKT07, OLAA03, MBLDP11, DASC93, MCLH05, CYH06, Hie04]. Para solucionar esta discrepancia se han estudiado diversas estrategias de finitización de protocolos [DKM+10, LMS07, VPP06]. Sin embargo, no hay resultados empíricos que den cuenta de su efectividad.

> *La hipótesis general de esta tesis es que nociones efectivas de cobertura pueden ser definidas en términos de abstracciones finitas definidas sobre el dominio semántico que describe un protocolo esperado de invocaciones.*

En esta tesis se proponen diferentes criterios de cobertura sobre abstracciones finitas de protocolos con espacio de estados infinito y se presentan experimentos cuyos resultados muestran que los criterios son buenos predictores de detección de fallas de conformidad de protocolos. Las implicaciones prácticas de este resultado pueden darse en contextos de desarrollo en los cuáles los test son escritos antes que las implementaciones. La generación de tests que busca la cobertura de las abstracciones propuestas tendría como resultado conjuntos de test con buena capacidad de detección de fallas de conformidad.

Los criterios de cobertura propuestos están definidos sobre *enabledness preserving abstractions* (EPAs) [dCBGU11]. Estas abstracciones cocientan un espacio de estados infinito en una cantidad finita de clases, cada una de las cuáles habilita un conjunto de acciones (o métodos) determinado. Por otro lado, los EPAs abstraen los parámetros de los métodos mediante eliminación existencial. Las transiciones entre los estados de un EPA (que corresponden a invocaciones de métodos que hacen evolucionar al programa de un estado a otro) son la clave de los criterios propuestos: mientras más transiciones sean cubiertas por un conjunto de tests, mayor es el nivel de adecuación del conjunto de tests de acuerdo al criterio.

Evaluación experimental

En este trabajo se ha evaluado la capacidad de detección de fallas de los criterios propuestos en cinco APIs de Java con relevancia industrial que poseen requerimientos no triviales en cuanto al orden de invocación de sus operaciones. Concretamente, se ha analizado la capacidad de los criterios para detectar mutantes usando conjuntos de tests generados al azar. Los experimentos realizados fueron diseñados para responder una serie de preguntas de investigación (RQs). Sucintamente, en esta tesis se estudia (RQ.1) la calidad un criterio de adecuación de testing basado en EPAs analizando la correlación entre cobertura de transiciones de EPAs y detección de fallas (RQ.1.1); y dado que la cobertura de transiciones está relacionada con el tamaño de los conjuntos de tests, el impacto de dicha cobertura para conjuntos de test de tamaño fijo (RQ.1.2). Luego se realiza un estudio cualitativo (RQ.2) con el objetivo de proporcionar un entendimiento más profundo sobre los resultados positivos obtenidos en RQ.1. Esto se ha hecho estudiando (i) la correlación entre la cobertura de transiciones de los EPAs y la cobertura de código alcanzada por los conjuntos de tests (RQ.2.1); (ii) si la partición del dominio implicitamente definida por las transiciones de los EPAs (un input i pertenece al subdominio definido por la transición t si y sólo si i cubre a t) tiende a generar subdominios densos en fallas (RQ.2.2); y (iii) la efectividad de las transiciones para revelar fallas (RQ.2.3). También se investiga si los resultados de RQ.1 y RQ.2 no son simplemente una consecuencia de haber obtenido cobertura de acciones (RQ.3). En tal sentido, se compara la correlación obtenida entre cobertura de acciones y detección de fallas con las correlaciones obtenidas para cobertura de transiciones (RQ.3.1), la densidad de los subdominios derivados de transiciones y acciones (RQ.3.2), y las efectividades de acciones y transiciones para revelar fallas (RQ.3.3). Como consecuencia de los resultados de RQ.3 se propone y estudia un criterio combinado para cobertura tanto de acciones como transiciones (RQ.4), investigando las correlaciones para detección de fallas de este nuevo criterio (RQ.4.1), y analizándolo para conjuntos de tests de tamaño fijo (RQ.4.2). Finalmente, se investiga la calidad de técnicas de priorización de casos de tests basados en cobertura de EPAs y se las compara con otras basadas en cobertura de código y órdenes aleatorios (RQ.5).

Sujeto a las amenazas de validez derivadas del modelo de fallas adoptado (que se discuten en profundidad en el Capítulo 9), los resultados de esta tesis sugieren que el criterio de adecuación de tests basado en la cobertura de transiciones de EPAs funciona bien como predictor de detección de mutantes (RQ.1.1). Los resultados también muestran que las correlaciones de este criterio con detección de mutantes son comparables (y mejores en algunos casos) que los que se obtienen de las coberturas de código (que son de caja blanca, y por lo tanto no pueden ser usados hasta tener el código de la aplicación disponible). También, para conjuntos de tests de tamaño fijo, se ve que aquellos con mejor adecuación al criterio son también mejores para la detección de fallas (RQ.1.2).

Los resultados de RQ.2 (cuyo proósito es tener un entendimiento más profundo de los fenómenos observados) refuerzan los resultados de RQ.1. Éstos indican que el criterio propuesto es un buen predictor de cobertura estructural (cobertura de sentencias y ramas) al ser aplicados sobre el código de los casos de estudio (RQ.2.1). Estos resultados implican que la construcción temprana de casos de test basados en la cobertura propuesta puede ser buena para obtener conjuntos de test con alta cobertura de código (que pueden ser extendidos una vez que el código se encuentre disponible).

Los resultados también sugieren que la partición del dominio implicitamente derivado de las transiciones de los EPAs tiende a producir subdominios que son densos en fallas (RQ.2.2). Esto es cercano a lo que se considera óptimo en el contexto de testing por particiones [JW89].

El estudio realizado para RQ.2.3, que investiga la efectividad de las transiciones para la detección de fallas, revela que en la mayoría de los casos para cada falla existe una transición altamente efectiva para detectarla ($> 90\%$ de efectividad), mientras que el resto de las transiciones suelen tener efectividades muy bajas ($< 10\%$). Más aún, los resultados muestran que la transición efectiva no siempre es la misma, sino que depende de cada falla. Esto está en concordancia con los resultados previos: dado que las fallas son inicialmente desconocidas, adquiere mucho sentido considerar aquellos conjuntos de tests que cubren más transiciones como más adecuados. Aquellos conjuntos de tests que alcancen el máximo nivel de adecuación deben cubrir todas las transiciones, y en consecuencia la transición altamente efectiva estará cubierta.

La experimentación referida a RQ.3 investiga si los resultados promisorios de cobertura de transiciones de los EPAs no se deben simplemente a que la cobertura de transiciones es un refinamiento de la cobertura de acciones. Los resultados de RQ.3.1 sugieren que para la correlación tanto con detección de fallas como con cobertura de código, la cobertura de transiciones es comparable (y mejor en varios casos) que la cobertura de acciones. Los resultados de RQ.3.2 indican que los subdominios definidos a partir de acciones son menos densos que aquellos derivados a partir de las transiciones. Finalmente, los resultados de RQ.3.3 muestran que las transiciones son, holgadamente, mucho más efectivas que las accionees para revelar fallas. Este conjunto de resultados sugiere que la cobertura de transiciones de los EPAs proveen ventajas sobre la cobertura de acciones, y justifican el estudio de un criterio combinado.

El conjunto de resultados correspondientes a RQ.4 muestra que considerar a ambos, acciones y transiciones, para medir el nivel de adecuación de un conjuto de tests genera resultados que mejoran los que se obtienen a partir tanto de accionese como de transiciones (consideradas individualmente) como predictor de fallas (RQ.4.1). Asimismo, los resultados muestran que al considerar conjuntos de tests de tamaño fijo, aquellos con mayor nivel de adecuación tienen mayor efectividad (RQ.4.2).

Finalmente, los resultados de RQ.5 muestran que la priorización de tests basadas en cobertura de EPAs es siempre mejor que aquellos basados en órdenes aleatorios, y que también tienden a alcanzar una alta tasa de detección de fallas a una velocidad similar (o más alta en algunos casos) a las de técnicas de priorización basadas en coberturas de caja blanca.

En conlusión, esta tesis da un pequeño paso para la comprensión de cómo la cobertura semántica puede ser utilizada para la detección de fallas de conformidad de protocolos. Los resultados sugieren que los criterios propuestos son buenos predictores de detección de fallas y de criterios de cobertura estructural clásica, como cobertura de sentencias y ramas. Por otro lado, las transiciones poseen una muy

interesante propiedad en relación a su efectividad para revelar fallas (mucho más alta que la de acciones) y, aprovechando este fenómeno, también se define un nuevo criterio que involucra cobertura tanto de acciones como transiciones. Las implicancias de los resultados son que los criterios de propuestos podrían ayudar a mejorar random testing, el desarrollo basado en tests, la selección de casos de test y, en general, técnicas de generación de casos de test a partir de especificaciones formales que vayan a ser a aplicadas sobre APIs que presentan prótocolos no triviales de invocación de sus acciones. Los resultados indican que todos estos enfoques podrían benficiarse al introducir heurísticas que busquen maximizar la cobertura de acciones y transiciones de los EPAs.

Contribuciones

Las Contribuciones de este trabajo pueden resumirse del siguiente modo:

1. utilización de abstracciones semánticas en el contexto de testing de conformidad de protocolos;
2. definición de criterios de adecuación de tests basados en EPAs;
3. una profunda evaluación experimental de los criterios propuestos que incluye:
   (a) un estudio de la calidad de los criterios como predictores de detección de fallas de conformidad de protocolos;
   (b) un estudio de la calidad de los criterios como predictores de cobertura de código:
   (c) un estudio de los criterios desde la perspectiva de partición en categorías;
   (d) un estudio de la efectividad de las transiciones para detectar fallas;
   (e) un estudio sobre las diferencias entre la cobertura de transiciones de EPAs y cobertura de acciones;
   (f) una comparación de los criterios propuestos con criterios basados en caja blanca y órdenes random en el contexto de priorización de casos de test.

Capítulo 2: Presentación del problema

En este capítulo se formalizan todas las nociones necesarias para la definición del problema abordado en esta tesis. Se define qué es el protocolo esperado de un artefacto de software, qué es el protocolo real de una implementación, cuál es la relación de conformidad que se espera que exista entre ellos y el criterio de adecuación basado en transiciones de EPAs.

Conformidad de protocolos

El término *conformidad de protocolos* ha sido extensamente utilizado abarcando diversos enfoques relacionados al chequeo de que una implementación se ajuste a una especificación. En este trabajo usamos un término más específico: protocolo de invocaciones de APIs. Como en el caso de verificación de typestates [BKA11, FGRY05, FYD+08, BA07, BBA09], el proposito final en este trabajo es la preservación de interoperabilidad: para satisfacer la relación de conformidad una API debe ser capaz de ejecutar todas las secuencias de operaciones que una cierta especificación define como legales. Como es usual en este contexto, se restringe el dominio del problema a especificaciones e implementaciones determinísticas con abstracción de resultados.

El *protocolo real* de una API puede ser definido como un sistema de transición etiquetado (LTS). Los estados del LTS son todas las configuraciones del estado interno del código. Si el código es una clase, entonces las configuraciones corresponden a todas las instancias estructuralmente diferentes. Si el código es una implementación de una API, las configuraciones son todas las posibles valuaciones de las variables internas de la implementación de la API. Las transiciones representan el efecto de una invocación exitosa de métodos con parámetros concretos. Una transición entre dos estados s y s' estará presente si y sólo si la ejecución del metodo asociado -con los parámentros anotados- en la configuración correspondiente a s eventualmente termina, no levanta ninguna excepción y modifica el estado interno del código dejándolo en el que se corresponde con s'. A modo de ejemplo, en la Figura 1 (a) se muestra un código erróneo de una clase `BoundedStack` (en el método `pop` la excepción debería levantarse únicamente si `top` es menor a 0), y en la Figura 1 (b) su protocolo real.

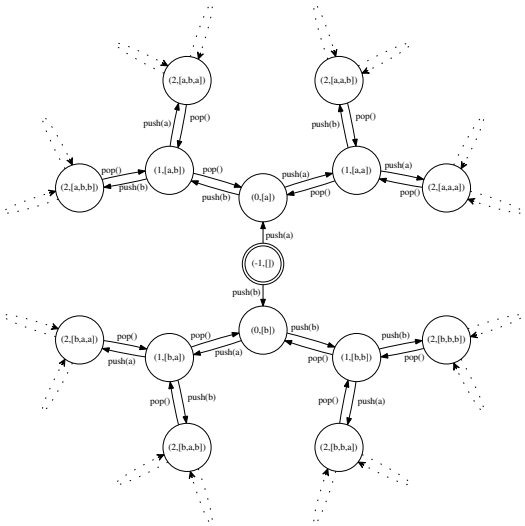
```
public class BoundedStack {
    Object[] elems;
    int top;

    public void BoundedStack() {
      elems = new Object[1000];
      top = -1;
    }

    public Object pop() {
        if (top <= 0) {
          throw new
              IllegalStateException();
        } else {
          return elems[top--];
        }
    }

    public void push(Object o) {...
}
```

(a)



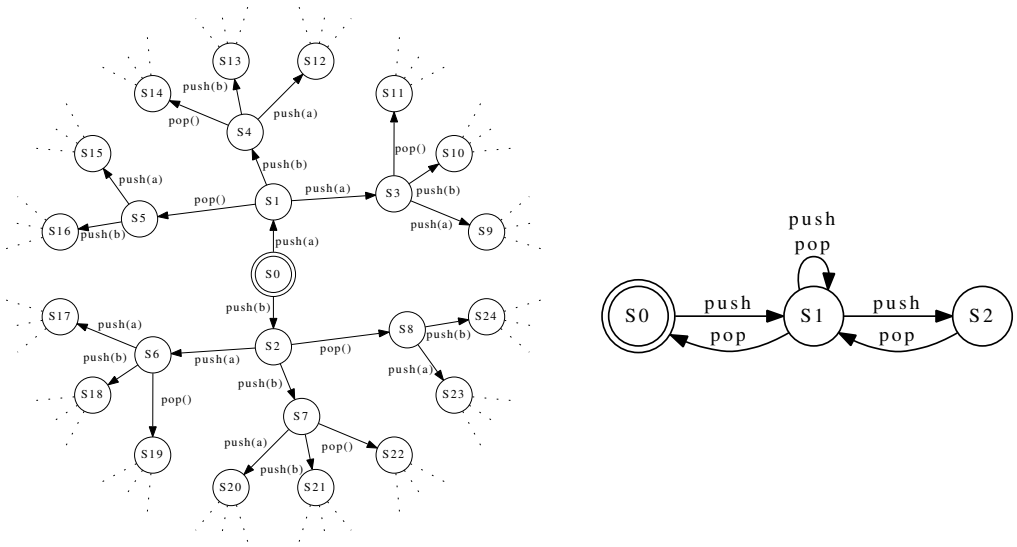
(b)

Figura 1: Fragmentos de (a) una implementación defectuosa de un `BoundedStack` y (b) su protocolo real

La semántica del *protocolo esperado* puede definirse en forma similar mediante un LTS que define cuál es el conjunto (potencialmente infinito) de secuencias de invocaciones válidas sobre un artefacto de código (cada invocación anotada con parámentros concretos). La figura 2 (a) muestra una porción del protocolo esperado de un `BoundedStack`. Una implementación estará en conformidad si su protocolo real acepta todas las secuencias que son legales de acuerdo al protocolo esperado. Una falla será entonces una secuencia de invocaciones que es aceptada por el protocolo esperado pero rechazada por el protocolo real. En el ejmplo del `BoundedStack` la secuencia $[push(a), push(a), pop(), pop()]$ es una falla, pues forma parte del protocolo esperado pero no es soportada por la implementación.

Enabledness preserving abstraction (EPA)

El objetivo de este trabajo es definir criterios de cobertura basados en la semántica de un protocolo esperado. Los trabajos clásicos de conformidad de protocolos se definen sobre máquinas de estado finitas [LY96]. Sin embargo, muchos protocolos



(a) (b)

Figura 2: (a) Fragmento del protocolo espardo de un `BoundedStack` y (b) su EPA

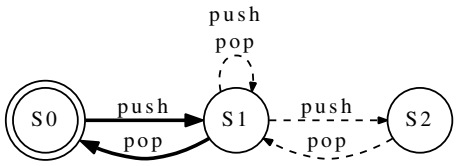
definen un espacio de estados infinito. La propuesta de la presente tesis es trabajar con *abstracciones finitas* del protocolo esperado. En particular, la abstracción que se utiliza es la enabledness preserving abstraction (EPA) presentada en [dCBGU11]. Básicamente un EPA es un LTS en el que las etiquetas son nombres de métodos (sin parámetros). Las EPAs abstraen el espacio de estados de un protocolo esperado cocientándolo de acuerdo a los métodos habilitados en cada estado. En otras palabras, dos estados de un LTS de un protocolo esperado son representados por el mismo estado del EPA si y sólo si en ambos estados se pueden ejecutar exactamente los mismos métodos (posiblemente usando distintos parámetros). La Figura 2 (b) muestra el EPA del protocolo esperado de la Figura 2 (a).

Criterio de cobertura de transiciones de EPAs

Comencemos por definir un *caso de test* como una secuencia de invocaciones de métodos con parámetros concretos y un *conjunto de tests* como un conjunto de casos de test. Nótese que el efecto de la ejecución de un caso de test sobre una instancia puede ser unívocamente interpretada como un camino en el LTS del protocolo esperado. A su vez, un camino en el LTS del protocolo puede ser univocamente simulado en un camino en el EPA. La Figura 3 muestra (a) un ejemplo de caso de test para `BoundedStack` y (b) la cobertura sobre el EPA que obtiene su ejecución. Notar que la implemetación defectuosa de la Figura 1 (a) fallará cada vez que se ejercite la transición $(S_1, \texttt{pop}, S_0)$. Se define entonces el nivel de adecuación de un conjunto de tests como el porcentaje de transiciones que son cubiertos por los casos test que lo componen. A modo de ejemplo, un conjunto de tests compuesto unicamente por el caso test de la Figura 3 (a) es 33, dado que sólo 2 de las 6 transiciones del EPA son cubiertas.

```
public void test() {
   BoundedStack bs = new BoundedStack();
   bs.push('a');
   bs.pop();
   assertEquals(0,size(bs));
}
```

(a)



(b)

Figura 3: (a) Ejemplo de un caso de test y (b) de su ejecución sobre el EPA

Capítulo 3: Diseño experimental

Resumen de los experimentos

El objetivo de los experimentos es medir la capacidad predictiva de detección de fallas del criterio de adecuación propuesto. Para poder responder las preguntas de investigación planteadas hace falta considerar algunas variables asociadas a los conjuntos de tests: *fallas detectadas*, *cobertura de código*, *cobertura de transiciones de EPAs* y *cobertura de acciones*. La estrategia experimental está basada en mutación de código del mismo modo que en [ABLN06, AZ03, BLW04].

La detección de fallas involucra (*i*) contar con el LTS del protocolo esperado y una implementación que satisfaga la relación de conformidad y (*ii*) obtener implementaciones que no satisfagan dicha relación. La estrategia usada en este trabajo consiste en seleccionar una implementación a ser usada como *implementación de referencia*. Dicha implementación es usada entonces como especificación y como implementación correcta del protocolo esperado. De esta manera, el protocolo real satisface la relación de conformidad por construcción.

La implementación de referencia también es usada como base para la generación de versiones que no satisfagan la relación de conformidad. Las versiones fallidas se obtienen aplicando operadores de mutación sobre la implementación de referencia. La identificación de fallas se realiza ejecutando casos de test tanto sobre la implementación de referencia como en las mutaciones. Cuando un mutante no consigue ejecutar una secuencia de llamadas que es válida de acuerdo a la implementación de referencia se considera que se ha matado a dicho mutante.

Casos de estudio

Para la experimentación se ha restringido el universo de potenciales casos de estudio a un único lenguaje de programación, de forma tal de permitir contar con una plataforma experimental uniforme en lo que respecta a mutación de código, generación de tests e infraestructura para detección de fallas. En particular se ha utilizado Java para aprovechar la disponibilidad de herramientas existentes y la dispondibilidad de clases que satisfacen el criterio de selección de casos de estudio utilizado: (*i*) código que presente un protocolo rico en término del orden en el cuál sus métodos deban ser invocados; (*ii*) código de relevancia industrial; y (*iii*) código para el cuál su EPA pueda ser obtenido. Siguiendo estos criterios hemos realizado los esutdios sobre las siguientes clases Java:

1. `Signature`: clase del JDK 1.4 que provee a las aplicaciones algoritmos para trabajar con firmas digitales;

2. `ListIterator`: clase del JDK 1.4 que provee la funcionalidad para iterar los

elementos guardados en una lista;

3. `Socket`: clase del JDK 1.4 que provee la funcionalidad para establecer conecciones TCP entre dos hosts;

4. `SMTPProcessor`: clase central de Java Email Server, responsable del procesamiento de las solicitudes recibidas;

5. `JDBCResultSet`: implementación de la interfaz `ResultSet` de la especificación JDBC de la base de datos HyperSQL 2.0.0.

Construcción de los EPAs de los LTSs del protocolo esperado

Los EPAs de cada caso de estudio fueron construidos a partir de las implementaciones de referencia, y dependiendo del caso de estudio se han utilizado distintas estrategias. Un recurso clave para la construcción fue la herramienta Contractor [dCBGU09], que construye EPAs tanto a partir de especificaciones basadas en contratos [dCBGU09] como a partir de código fuente [dCBGU11]. Los modelos producidos por Contractor sobreaproximan los EPAs, ya que pueden incluir transiciones espurias.

Los EPAs del protocolo esperado de `Signature`, `ListIterator`, `Socket` y `SMTPProcessor` fueron generados usando Contractor sobre código fuente [dCBGU11]. En ninguno de estos casos los modelos generados contuvieron transiciones espurias, con lo que los EPAs construidos son los del protocolo real de las implementaciones de referencia. En el caso de `JDBCResultSet` el EPA fue obtenido usando Contractor sobre la especificación mediante contratos definida en [BBA09] de la interface `ResultSet` de JDBC. Esta especificación se ha validado comparando las precondiciones de cada método contra las guardas de las sentencias que levantan excepciones en la implementación de referencia. El análisis realizado permite concluir que el EPA generado es efectivamente el de la implementación de referencia. La figura 4 muestra el EPA generado para la clase `Socket`.

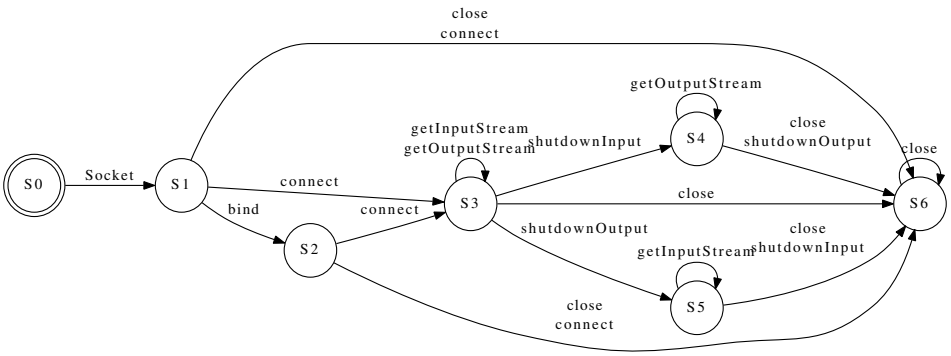


Figura 4: EPA de la semántica del protocolo esperado de `Socket`

Detalles de la implementación del experimento

Para cada caso de estudio se han generado 10000 casos de test usando Randoop [PLEB07], una herramienta de generación automática de casos de test para Java. Randoop genera secuencias de invoaciones seleccionando al azar entre los métodos públicos de una clase. Como las secuencias generadas no necesariamente son válidas de aucerdo al protocolo esperado de la clase, se instrumentó el código usando AspectJ para garantizar que éstas no caigan fuera del protocolo.

Para la obtención de mutantes se ha utilizado μ-Java [MOK05], un sistema de generación de mutantes para clases Java. Con el objetivo de obtener tantas implementaciones fallidas como fuera posible, se han usado todos los operadores de mutación. Algunos de los mutantes generados podrían ser semánticamente equivalentes a la implementación de referencia. El problema de detección de mutantes equivalentes se ha probado indecidible [OP97], y hacer inspección manual no es posible debido a la gran cantidad de mutantes generados (además de ser un práctica propensa a errores). Aquellos mutantes no descubiertos por ningún caso de test fueron considerados correctos del mismo modo que en [ABLN06].

Para medir la cobertura de código se ha usado la herramienta Cobertura. Ésta instrumenta el código y mantiene registro de las sentencias que son alcanzadas a medida que se ejecutan los casos de test. Reporta sobre cobertura de sentencias y ramas.

El Cuadro 1 resume la información de cada uno de los casos de estudio.

Clase	LOC	Casos de Test	Mutantes		EPAs		
			Total	Muertos	Estados	Transiciones	
						Total	Cubiertas
`ListIterator`	59	10000	207	145 (70 %)	8	68	63 (93 %)
`Signature`	121	10000	150	96 (64 %)	4	29	27 (93 %)
`Socket`	144	10000	98	59 (60 %)	7	20	18 (90 %)
`SMTPProcessor`	404	10000	570	317 (56 %)	12	85	70 (82 %)
`JDBCResultSet`	785	10000	404	259 (64 %)	9	247	199 (81 %)

Cuadro 1: Información de las clases utilizadas

Capítulo 4: Análisis cuantitativo

En este capítulo se reporta el análisis realizado para responder la primera pregunta de investigación:

> **RQ.1.** ¿Hasta qué punto el nivel de adecuación de un conjunto de tests al criterio de cobertura de transiciones de EPAs predice la capacidad de detección de fallas de conformidad de protocolos?

El criterio como predictor de fallas

La pregunta RQ.1 ha sido refinada en dos preguntas más específicas, siendo la primera de ellas la siguiente:

> **RQ.1.1.** ¿El nivel de adecuación de un conjunto de tests al criterio de cobertura de transiciones de EPAs tiene alta correlación con detección de fallas de conformidad de protocolos?

Para poder responder RQ.1.1 se ha calcualdo el coeficiente de correlación no paramétrico de Spearman ρ, que mide hasta que punto la relación entre dos variables puede ser descripta mediante una función monótona. El coeficiente arroja valores en el intevalo real [-1,1]. Valores cercanos a -1 y 1 indican una fuerte dependencia entre las variables (inversa y directa respectivamente), mientras que valores cercanos a 0 indican ausencia de dependencia. También se ha calculado el coeficiente entre

cobertura de código y detección de fallas. Los resutados pueden observarse en el Cuadro 2

Clase	Sentencias	Ramas	Transiciones
`ListIterator`	0.48	0.61	**0.84**
`Signature`	0.65	0.71	**0.73**
`Socket`	0.81	**0.86**	0.72
`SMTPProcessor`	0.66	**0.78**	0.69
`JDBCResultSet`	0.89	**0.92**	0.68

Cuadro 2: Correlación entre cobertura y detección de fallas. El valor más alto de una fila se indica en negrita.

Los resultados sugieren que el nivel de adecuación a la cobertura de transiciones de EPAs es un buen predictor para la detección de mutantes, con correlaciones entre 0.68 y 0.84 dependiendo del caso de estudio. Los resultados muestran que las correlaciones son comparables o mejores que aquellas obtenidas para criterios de caja blanca. En todos los casos de estudio la cobertura de transiciones funciona mejor que la cobertura de sentencias, y en dos casos de estudio también funciona mejor que la cobertura de ramas. Por otro lado, los valores obtenidos en el caso de transiciones son más estables que los de los otros criterios.

Test de hipótesis para conjuntos de test de tamaño fijo

La segunda pregunta más específica para RQ.1 aborda el problema de la influencia del tamaño de un conjunto de test en su eficacia:

> **RQ.1.2.** ¿Son mejores los conjuntos de test que tienen mayor nivel de adecuación al criterio de cobertura de transiciones de EPAs en términos de detección de fallas de conformidad de protocolos al fijar el tamaño de los conjuntos?

El impacto del tamaño en la efectividad de los conjuntos de tests ha sido abordado en diversos estudioss [NA09, AGWX08]. Más allá de que el tamaño no es el único factor que determina la calidad de un conjunto de tests, es esperable que mientras mayor sea éste (medido como número de invocaciones sobre el sistema testeado), mayor será su capacidad de detectar fallas [NA09]. Los tests que alcanzan mayores cobertura en las EPAs son, generalmente, también más grandes. Para descartar que el beneficio de cubrir más transiciones sea simplemente derivado del tamaño de los conjuntos de test se analizan los resultados del siguiente modo: dado un tamaño de conjunto de test se quiere determinar si los conjuntos con mayor cobertura de transiciones tienden a descubrir más fallas.

Las muestras para cada tamaño no son lo suficientemente grandes como para poder obtener resultados estadísticamente significativos. Por lo tanto, en cada caso de estudio se divide a los conjuntos de test en 10 grupos de tamaño similar. De este modo, la diferencia de tamaño entre los individuos de cada grupo no excede el 10% de la diferencia entre el conjunto de tests más grande y el más pequeño. Se quiere mostrar que dado un grupo, al seleccionar un conjunto de tests entre aquellos con mayor cobertura va a tener mejor capacidad de detección de fallas que si se lo selecciona del resto de los conjuntos. Dado un grupo, se define a los conjuntos de alta cobertura como aquellos que alcanzan al menos un 80% del conjunto con mayor cobertura de ese grupo.

Para comparar a los conjuntos de tests de alta y baja cobertura de cada grupo se ha utilizado el test de hipótesis no paramétrico U Mann-Whitney, que evalua si la probabilidad de que un individuo de un grupo exceda a un individuo del otro no sea igual a 0.5. Bajo la hipótesis nula, indiviudos de ambos grupos cuentan con misma probabildiad de exceder uno al otro, mientras que la hipótesis alternativa esta probabilidad no es de 0.5. Se rechaza la hipótesis nula cuándo el p-valor resultante del test es menor a 0.05. En el caso del presente experimento, rechazar la hipótesis nula indica que los conjuntos de test de alta cobertura son mejores que los de baja cobertura en términos de detección de fallas de conformidad de protocolos, y que ésto no es producto del azar. Para medir la magnitud de la mejora se ha calculado tamaño del impacto A_{12} de Vargha Delaney (ES). En cada grupo, A_{12} es un estimador de la probabildiad de que al elegir un conjunto de tests de alta cobertura se detecten más fallas que al elegirlo del resto de los tests. También se reporta un intervalo de confianza para A_{12} fijado en un nivel de confianza del 95%. Los resultados pueden verse en la Tabla 4.2 del Capítulo 4.

Los resultados sugieren que dado un tamaño de conjunto de tests, aquellos que más transiciones cubren también detectan más fallas de conformidad. Contando todos los grupos de los cinco casos de estudio, se ha encontrado evidencia estadística de la ventaja de cubrir más transiciones en 39 de los 50 grupos. En general, aquellos grupos para los que no se ha encontrado evidencia corresponden a conjuntos de test de gran tamaño. Como es esperable, el impacto de la cobertura de transiciones disminuye a medida que el tamaño de los conjuntos de test aumenta.

Capítulo 5: Análisis cualitativo

En este capítulo se estudia la segunda pregunta de investigación, cuyo proposito es comprender de manera más profunda los resulados de RQ.1.

> **RQ.2.** ¿Por qué el nivel de adecuación de un conjunto de tests a la cobertura de transiciones de EPAs predice la capacidad de detección de fallas de conformidad de protocolos?

Como en el caso de RQ.1, también RQ.2 es refinada en distintas preguntas.

El criterio como predictor de cobertura de código

La primera de las preguntas derivadas de RQ.2 analiza la capacidad del criterio de cobertura de transiciones de predecir la cobertura de código que se alcanza sobre las implementaciones.

> **RQ.2.1.** ¿El criterio de cobertura de transiciones de EPA predice la cobertura de sentencias y ramas?

Para responder esta pregunta, en primer lugar se calcula nuevamente el coeficiente de correlación de Spearman. Los resultados pueden observarse en el Cuadro 3.

Los resultados parecen indicar que el criterio de cobertura de transiciones de EPA es razonablemente bueno como predictor de cobertura de sentencias y ramas. Las correlaciones contra cobertura de ramas es moderada en un caso de estudio y alta en los cuatro restantes. Asimismo los valores presentan estabilidad, cercanos a 0.7 en todos los casos. En cuatro de los cinco casos de estudio la correlación con

Clase	Transiciones y sentencias	Transiciones y ramas
`ListIterator`	0.66	**0.70**
`Signature`	0.63	**0.70**
`Socket`	0.77	**0.77**
`SMTPProcessor`	0.70	**0.73**
`JDBCResultSet`	**0.67**	0.66

Cuadro 3: Correlación entre transiciones de EPA y cobertura estrucutral. El valor más alto de una fila se indica en negrita.

cobertura de ramas es mayor que la que se obtiene con cobertura de sentencias. En el tipo de software analizado la selección de ramas (que depende de la evaluación de sentencias condicionales) está frecuentemente basada en el estado interno de un objeto. Los altos valores hallados indican que los estados de los EPAs capturan efectivamente propiedades importantes del estado de los objetos concretos.

También en este caso se analiza qué sucede con los conjuntos de tests cuando son agrupados de acuerdo a su tamaño, usando los mismos grupos utilizados en RQ.1.2. Se realizaron tests de hipótesis U Mann-Whitney rechazando la hipótesis nula cuándo el p-valor obtenido del test es menor a 0.05. En este caso rechazar la hipótesis nula indica que los test de alta cobertura de transiciones de EPA tienen también mayor cobertura de código. Nuevamente se ha medido el tamaño del impacto calculando A_{12} para estimar la probabilidad de que al elegir un conjunto de tests de los de alta cobertura se obtenga uno que cubre más código. También se calculó un intervalo de confianza para dicho valor fijando el nivel de confianza en 95%. Los resultados pueden verse en el Cuadro 5.2 del Capítulo 5.

Considerando los 5 casos de estudio, existe evidencia de la ventaja de los conjuntos de tests de alta cobertura en 34 de los 50 grupos en relación a la cobertura de sentencias. En lo que respecta a cobertura de ramas, la evidencia existe en 43 de los 50 grupos. En definitiva, en la gran mayoría de los casos existe evidencia de la ventaja de unos sobre los otros.

Una perspectiva de partición en categorías

La segunda pregunta derivada de RQ.2 analiza los resultados desde una perspectiva de partición en categorías del dominio de entradas (*inputs*).

> **RQ.2.2.** ¿La partición en categorías implícitamente definida par la cobertura de transiciones de EPA genera subdominios con alta tasa de fallas?

La partición en categorías se refiere una familia general de estrategias de testing [JW89]. Esta consiste en dividir el dominio de inputs de un programa en subdominios de forma tal que entre todos los subdominios se cubra todo el espacio de inputs. Luego el programa es testeado seleccionando inputs de cada subdominio. Idealmente, cada subdominio debería contener o bien sólo inputs que hagan fallar al programa o bien sólo inputs para los que el programa funcione correctamente. Dado que los errores son a priori desconocidos es difícil alcanzar este ideal. De todas formas, si existe un subdominio con alta tasa de fallas (es decir, que gran parte de los inputs que contiene revelen que el programa falla), la estrategia resulta efectiva [JW89].

El criterio de cobertura propuesto divide el dominio de inputs mediante las transiciones de los EPAs. Un caso de test c pertenece a la categoría definida por una

transición t si y sólo si la ejecución de c ejercita la transición t. Nótese que los subdominios generados no forman conjuntos disjuntos. La Figura 5.1 del Capítulo 5 muestra las tasas de falla de los subdominios más densos de cada versión que no satisface la relación de conformidad y la tasa de fallas del dominio completo. Los valores numéricos pueden observarse en el Cuadro 5.4 del Capítulo 5. El mismo análisis se realizó tomando sólamente en cuenta aquellos mutantes que resultaron más díficles de detectar (aquellos que fueron detectados por menos del 20% de los tests). Los resultados pueden verse en las Figura 5.2 y el cuadro 5.5 del Capítulo 5.

Los resultados indican que la partición del dominio derivada implicitamente de las transiciones de un EPA tienden a producir subdominios densos en fallas. Es decir, subdominios con alta tasa de fallas. Más específicamente, en todos los casos de estudio para el 50% de las versiones no conformantes existen subdominios tales que el 100% de sus inputs los revelan como fallidos. Asimismo, para el 75% de las versiones no conformantes la partición de dominios realizada genera tasas de fallas que son al menos seis veces mayores que las del dominio completo. Al mirar únicamente las versiones más difíciles de detectar, para el 50% de las versiones que no satisfacen la relación de conformidad sus tasas de falla son al menos dos veces mayor (llegando en algunos casos a mejoras de dos órdenes de magnitud) que los del dominio completo.

Efectividad de las transiciones

La última pregunta derivada de RQ.2 es la siguiente:

> **RQ.2.3.** ¿Las transiciones son altamente efectivas para detectar implementaciones que no satisfacen la relación de conformidad?

A partir de RQ.2.2 resulta tentador explicar el fenómeno observado del siguiente modo: dada una implementación fallida I, existe una transición del EPA que al ser ejercitada expone a I como no conformante. Sin embargo, la existencia de subdominios densos en fallas no necesariamente implica la existencia de transiciones efectivas. Por ejemplo, una transición podría exponer una versión fallida sólo después de haber sido ejercitada varias veces. Más aún, un caso de test perteneciente al subdominio de una transición t podría revelar a I como fallida al atravesar una transición que no sea t. Esto significa que el análisis de partición de subdominios es de grano demasiado grueso. En RQ.2.3 se analiza la efectividad de las transiciones más que de los subdominios generados a partir de ellas.

Dada una implementación I y una transición t se define la efectividad de t para I como el cociente entre la cantidad de veces que que I es expuesta como fallida y la cantidad de veces que I ejercita t. Para comprender las efectividades de las transiciones (i) se divide las efectividades en 10 grupos correspondientes a valores en los intervales [0,0.1],(0.1,0.2],...,(0.9,1.0] y (ii) para cada implementación no conformante se cuenta el porcentaje de transiciones cuyas efectividades cae en cada intervalo de efectividades. Los boxplots de la Figura 5.3 del Capítulo 5 muestran la distribución de las efectividades en los distintos intervalos para cada caso de estudio.

Algo similar ocurre en todos los casos de estudio: dada una implementación fallida, casi todas las transiciones tienen baja efectividad ($< 10\%$), unas pocas tienen muy alta efectividad ($> 90\%$), y en el medio no hay prácticamente ninguna. Una pregunta clave es si la transición altamente efectiva es siempre la misma o varía de acuerdo a cada implementación no conformante. Las Figuras 5.4 y 5.5 del Capítulo 5 muestran, para cada transición, el porcentaje de versiones fallidas para las cuáles

resultan de alta efectividad. Como puede observarse, cuál es la transición con alta efectividad depende de cada implementación fallida. La existencia de transiciones con alta efectividad refuerza el criterio propuesto. Mientras mayor sea el nivel de adecuación de un conjunto de tests, mayor es también la probabilidad de que dicho conjunto ejercite la transición altamente efectiva.

Capítulo 6: Comparación entre transiciones y acciones

Cada transición de un EPA está asociada con una acción en una relación de muchos-a-uno. Por ejemplo, el el EPA de la clase `Socket` de la Figura 4 hay dos transiciones correspondientes al método `getOutputStream`. La diferencia es que mientras que en el estado S_3 la invocación a `getInputStream` es legal, dicha invocación desde S_4 resulta en una violación del protocolo. El propósito de la siguiente pregunta de investigación es entender si las observaciones hechas sobre las transiciones de los EPAs son una consecuencia de los resultados que se obtienen usando directamente acciones.

RQ.3. ¿Cómo resultan los resultados correspondientes a las preguntas anteriores cuándo se considera cobertura de acciones en lugar de cobertura de transiciones de EPA?

Al igual que con las preguntas anteriores, RQ.3 también ha sido refinada en distintas preguntas más específicas.

Cobertura de acciones

La primera pregunta derivada de RQ.3 es la siguiente:

RQ.3.1. ¿Cómo resulta la cobertura de acciones como predictor de fallas y cobertura de código al compararla con cobertura de transiciones de EPA?

Para analizar esta pregunta tambien se ha calculado el coeficiente de correlación de Spearman. El Cuadro 4 muestra los coeficientes de de cobertura de acciones y de cobertura de transiciones para detección de fallas y cobertura estructural de código.

Clase	Detección de fallas		Cobertura sentencias		Cobertura ramas	
	Transición	Acción	Transición	Acción	Transición	Acción
`ListIterator`	**0.84**	0.48	0.66	**1.00**	**0.70**	0.58
`Signature`	**0.73**	0.68	0.63	**0.79**	0.70	**0.75**
`Socket`	0.72	**0.76**	0.77	**0.93**	0.77	**0.89**
`JDBCResultSet`	0.68	**0.77**	**0.67**	0.60	**0.66**	0.57
`SMTPProcessor`	**0.69**	0.55	0.70	**0.92**	0.73	**0.76**

Cuadro 4: Correlación entre transiciones y acciones con detección de fallas y cobertura estructural

Respecto a la detección de implementaciones que no satisfacen la relación de conformidad, la cobertura de transiciones de EPA obtiene mejores coeficientes de correlación en tres de los cinco casos de estudio. Adicionalmente, los valores asociados a la cobertura de transiciones son también más estables. En términos de cobertura estructural de código, la cobertura de acciones obtiene correlaciones más

altas. Esto es esperable porque la correlación mide hasta qué punto la relación entre dos variables puede describirse usando una función monótona. En el caso de acciones y cobertura de código, esto significa medir hasta qué punto es cierto que cubrir más acciones implica ejercitar más código. Cada vez que una acción no ejecutada previamente es ejecutada se ejercita una porción de códgio no ejercitada hasta ese momento. Esta observación no es cierta en el caso de la cobertura de transiciones, dado que cubrir dos transiciones asociadas a una misma acción no necesariamente aumenta el código ejercitado.

Acciones vs. transiciones como partición en categorías

El objetivo de la siguiente pregunta derivada de RQ.3 es comparar las particiones de dominio resultantes de los criterios de cobertura de acciones y cobertura de transiciones de EPA:

> **RQ.3.2** ¿Cómo resultan las tasas de falla de los subdominios derivados de la cobertura de acciones cuándo se las comprara con la de los subdominios derivados de la cobertura de transiciones de EPA?

Para responder esta pregunta se analizan las tasas de falla de los subdominios derivados de la cobertura de acciones del mismo modo en el que se hizo con la cobertura de transiciones. Se considera que un caso de test c pertenece al subdominio definido por la acción a si y sólo si a es invocada en c. La Figura 6.1 y el Cuadro 6.2 del Capítulo 6 muestran los resultados obtenidos.

Mientras que el el caso de los subdominios derivados de las transiciones de los EPA hay subdominios con tasa de falla 1 para el 50% de las implementaciones fallidas, en el caso de los subdominios derivados de las acciones esto sólo puede afirmarse para un 25% de estas implementaciones. Detodas formas, las tasas de falla para los subdominios de acciones son aceptables: para el 50% de las implementciones fallidas alcalzan entre 0.78 y 1, dependiendo del caso de estudio.

También en este caso se restringió el universo de mutantes a aquellos que fueron detectados por menos del 20% de los casos de test. Los resultados pueden verse en la Figura 6.2 y en el Cuadro 6.3 del Capítulo 6. En este caso, los valores alcanzados tanto a partir de las acciones como de las transiciones de EPAs no presentan grandes diferencias.

Efectividad de acciones vs. efectividad de transiciones

Para comparar acciones y transiciones, se puede hacer una comparación más precisa si en lugar de considerar los subdominios derivados se analizan las efectividades. La última pregunta derivada de RQ.3 tiene como objetivo compara las efectividades de las acciones al compararla con la de las transiciones de EPAs.

> **RQ.3.3** ¿Cómo resulta la efectividad de las acciones en relación a la de las transiciones para la detección de fallas de conformidad?

Al igual que con las transiciones, la efectividad de una acción a para una implementación fallida I se define como el cociente entre la cantidad de veces que al ejecutar una acción se revela I como fallida y la cantidad de vece que a es ejecutada.

Es importante notar que hay dos situaciones en las cuales la efectividad de una transición no puede ser mayor que la de una acción: (i) cuándo una acción tiene

una única transición asociada y (*ii*) cuándo una acción tiene efectividad 1 para una determinada implementación. De todas formas existe una gran cantidad de implementaciones no conformantes para las que no se da ninguna de estas condiciones (ver Sección 6.3).

A continuación se aborda qué sucede en los casos en los que sí existe posibilidad de mejora. La Figura 6.3 muestra los porcentajes de mejora de las transiciones sobre las acciones de cada uno de los casos de estudio. Los valores númericos pueden verse en el Cuadro 6.5.

El nivel de mejora en la efectividad de las transiciones sobre las acciones varía significativamente entre los distintos casos de estudio. Sin embargo, en todos los casos las mejoras son muy altas para una gran cantidad de implementaciones que no satisfacen la relación de conformidad. En promedio las mejoras van del 81% al 708.52% dependiendo del caso de estudio. Se puede concluir de estos datos que las transiciones resultan mucho más efectivas que las acciones. Esta observación refuerza la idea de cubrir transiciones (en lugar de simplemente ejecutar acciones) para revelar implementaciones fallidas.

Capítulo 7: Combinando transiciones y acciones

Los resultados del Capítulo 6 indican que puede haber una opurtinadad de conseguir un criterio efectivo si se combinan las transiciones de EPAs y las acciones para definir un criterio de cobertura. En este capítulo se define y estudia un criterio combinado para responder la siguiente pregunta de investigación.

> **RQ.4.** ¿Cómo son los resultados de RQ.1 y RQ.2 al adoptar un criterio de cobertura definido tanto en función de acciones como transiciones de EPA?

También esta pregunta es refinada en distintas subpreguntas que se presentan en las próximas secciones.

Un criterio combinado

El criterio combinado involucra el cubrimiento tanto de acciones como de transiciones del EPA. La idea subyacente es que si existe una acción que se encuentra habilitada y todavía no ha sido ejecutado, seleccionarla. Cubrir una nueva acción aumenta la cobertura de código y adicionalmente implica ejercitar una transición todavía no ejercitada. Cúndo todas las acciones habilitadas en un determinado estado ya han sido ejecutadas, el criterio prioriza ejercitar una transición todavía no ejercitada. Más formalmente, el nivel de adecuación dado por este criterio está dado por un par ordenado (j, k), donde j representa el porcentaje de acciones invocadas y k el porcentaje de transiciones ejercitadas. Para poder medir la efectividad del criterio para la detección de fallas es necesario establecer un orden de adecuación entre los conjuntos de tests, lo cuál se define de forma lexicográfica, dando prioridad a las accionese por sobre las transiciones.

El criterio como predictor de detección de fallas y cobertura estructural

La primera pregunta derivada de RQ.4 analiza la capacidad predictiva del criterio combinado tanto para la detección de fallas de conformidad como de cobertura estructural sobre código.

RQ.4.1. ¿Cómo es la correlación entre la adecuación del criterio combinado comparada con la cobertura de acciones y la cobertura de transiciones de EPA?

Nuevamente en este caso se calcula el coeficiente de correlación de Spearman. Los resultados pueden observarse en el Cuadro 5.

Clase	Detección de Fallas			Cob. Sentencias			Cob. Ramas		
	A+T	T	A	A+T	T	A	A+T	T	A
`ListIterator`	**0.84**	**0.84**	0.48	0.71	0.66	**1.00**	0.50	**0.70**	0.58
`Signature`	**0.73**	**0.73**	0.68	0.78	0.63	**0.79**	**0.79**	0.70	0.75
`Socket`	0.75	0.72	**0.76**	0.74	0.77	**0.93**	0.70	0.77	**0.89**
`JDBCResultSet`	**0.80**	0.68	0.77	**0.78**	0.67	0.60	**0.91**	0.66	0.57
`SMTPProcessor`	**0.69**	**0.69**	0.55	0.69	0.70	**0.92**	0.66	0.73	**0.76**

Cuadro 5: Correlación entre coberturas del criterio combinado, de transiciones y de acciones con detección de fallas y cobertura estructural

Los resutlados muestran que excepto en un caso, las correlaciones más altas entre cobertura y detección de fallas son las del criterio combinado. En términos de cobertura estructural, la cobertura de acciones sigue predominando.

Test de hipótesis para conjuntos de test de tamaño fijo

La segunda pregunta derivada de RQ.4 indaga sobre la conveniencia del criterio combinado al fijar los tamaños de los conjuntos de test.

RQ.4.2. ¿Son mejores los conjuntos de test que tienen mayor nivel de adecuación al criterio combinado en términos de detección de fallas y cobertura estructural al fijar el tamaño de los conjuntos de test?

También en este caso se han realizado los test U de Mann-Whitney, se ha calculado el tamñao del efecto A_{12} de Vargha y Delaney y un intervalo de confianza para A_{12} fijado en un nivel de confianza del 95%. Se ha utilizado la misma división en grupos de tests (de acuerdo a su tamaño) usados en el Capítulos 4 y 5. Dado que en este caso el nivel de adecuación de un test está dado por un par ordenado, no se puede aplicar el mismo criterio de considerar de alta cobertura a aquellos que alcanzan un 80% de la cobertura máxima del grupo. En este caso, cada grupo, se ha dividido entre el 20% de mayor adecuación como los test con alta adecuación y el restante 80% como los de baja adecuación.

El cuadro 7.2 del Capítulo 7 muestra los resultados de los tests. Como puede observarse, en 44 de los 50 grupos de test existe evidencia estadística de que aquellos test más adecuados de acuerdo al criterio combinado son mejores que el resto. Estos resultados son mejores que al considerar únicamente las transiciones de EPAs. Por otro lado, al considerar la cobertura de código puede verse que en 40 y 48 de los grupos hay evidencia estadística de la conveniencia de los más adecuados para cobertura de sentencias y ramas respectivamente.

Capítulo 8: Análisis de priorización de casos de test

En este capítulo se aborda la última pregunta de investigación, cuyo objetivo es analizar la calidad de técnicas de priorización de casos de test basadas en cobertura de EPAs.

RQ.5. ¿Cómo son las tasas de detección de fallas de técnicas de priorización basadas en cubrimiento de EPAs en relación a aquellas basadas en cubrimiento de código y órdenes aleatorios?

Priorización de casos de test

El problema de priorización de casos de test consiste en ordenar los casos de test de forma tal que las fallas sean detectadas lo antes posible. En [EMR00] los autores presentaron una métrica para medir la eficiencia de una técnica de priorización que desde entonces se ha transformado en un estándar. Esta métrica mide la velocidad de detección de fallas de un conjuto de tests calculando el *Porcentaje Promedio de Fallas Detectadas* (APFD) durante el ciclo de vida del conjunto de tests. El APFD es un valor entre 0 y 100, y mientras más rápida es la detección de fallas, mayor es el valor.

Técnicas de priorización basadas en cobertura

La priorización basada en cobertura se basa en la hipótesis de que mientras antes se logre coberutra (de acuerdo a alguna métrica), antes también se encontrarán más fallas. Por lo tanto, estas técnicas ordenan los casos de test para maximizar lo antes posible medidas de cobertura (como por ejemplo cobertura de código, de requerimientos, etcétera). Diversas técnicas de priorización han sido propuestas y evaluadas en la literatura [YH12]. Éstas pueden ser divididas en dos grandes grupos: (*i*) por un lado las que ordenan teniendo en cuenta la cobertura individual de cada caso de test y (*ii*) las que ordenan basandose en la cobertura adicional que agregan los casos de test a la cobertura obtenida hasta ese momento. Experimentos recientes han provisto evidencia de que las técnicas de cobertura adicional producen mejores órdenes que los resultantes de la cobertura individual [NABL13]. Por lo tanto, este capítulo está centrado en las siguientes técnicas de cobertura adicional:

- Cobertura adicional de transiciones de EPA
- Cobertura adicional de acciones y transiciones de EPA (criterio combinado)
- Cobertura adicional de ramas
- Cobertura adicional de sentencias

Por otro lado, también se calculan los APFD de órdenes aleatorios.

Resultados

La Figura 8.3 del Capítulo 8 los valores APFD obtenidos por cada técnica en cada uno de los cinco casos de estudio. Los valores numéricos pueden observarse en el Cuadro 8.2. Los valores de APFD basados en técnicas de cobertura adicional son mayores que los que se obtienen con órdenes aleatorios en todos los casos. Por otro lado, también puede observarse que en general los valores APFD de órdenes basados en coberutra de EPAs son mayores o muy similares a los que se obtienen al priorizar cobertura de código.

Al igual que en capítulos anteriores, también se ha calculado los valores APFD restringiendo el universo de implementaciones fallidas a aquellas que son más difíciles de detectar (es decir, detectadas por menos del 20% de los casos de test). Los resultados pueden verse en la Figura 8.4 y el Cuadro 8.3. Al restringir las implementaciones

consideradas, todos los valores más altos excepto uno corresponden a criterios que priorizan la cobertura de EPAs.

Capítulo 9: Amenazas de validez

Los resultados presentados en esta tesis están sujetos a algunas amenazas de validez. En este capítulo se discuten, distinguiendo entre amenazas internas, externas y de construcción.

Amenazas de validez externas

Estas amenazas se refieren a la posibilidad de generalizar los resultados presentados. Por un lado, la propuesta presentada no es válida para software que no tenga restricciones en cuanto al orden en el que sus acciones deban ser invocadas. Aún si sólo se considerara software con protocolos ricos, el estudio presentado fue sólo hecho sobre cinco casos de estudio que podrían no ser suficientemente representativos. Sin embargo, más allá de estas debilidades, el enfoque y los casos de estudios presentados son de relevancia para la comunidad de verificación de typestates [BKA11, FGRY05, FYD+08, BA07] y sistemas concurrentes [Sch99, MK06].

Otra amenza resulta de que en algunos casos de estudio la cobertura alcanzada sobre los EPAs no es lo suficientemente alta. Esto se debe principalmente a haber utilizado una generación random de casos de test para los cuáles alcanzar estados "profundos" de los EPA es dificultoso. Mediante una generación guiada de casos de test se podrían haber alcanzado mayor cobertura, pero sin dudas también habría aumentado mucho la posibilidad de que los conjuntos de test fuesen sesgados.

Amenazas de validez internas

Las amenazas de validez interna aparecen como consecuencia de cómo se han realizado los experimentos. Una amenaza importante es que los EPAs utilizados sean efectivamente los EPAs de los LTSs de los protocolos de los casos de estudio. El riesgo de haber usado abstracciones que no sean las abstracciones correctas está mitigado por el proceso de construcción sistemático utilizado, la validación con modelos construidos por terceros y las inspecciones manuales realizadas.

Otra amenaza se desprende de la forma que se han generado implementaciones fallidas (usando una herramienta de mutación). Si el análisis basado en mutantes es un modelo de errores realísta o no sigue siendo tema de debate en la comunidad de ingeniería de software, pero está más allá del alcance de esta tesis. De todas formas, algunos trabajos recientes han mostrado evidencia estadística de la su validez (ver por ejemplo [XXTdH11]), y es una forma muy utilizada para evaluar técnicas de testing.

Otra amenza surge como consecuencia del modelo de fallas adoptado. Se pueden hacer cálculos incorrectos y que su manifestación se produzca tiempo después. Este es un problema para todos los trabajos que usan casos de test, y no algo particular de los experimentos descriptos en esta tesis. Esta amenaza se ha mitigado usando casos de test largos.

Por último, las amenazas que se desprenden de la infrastructura utilizada durante los experimentos es menor, pues se han usado herramientas estándar como μ-Java y Randoop y técnicas simples de instrumentación usando AspectJ para registrar la cobertura de los tests sobre los EPAs.

Amenazas de construcción

Una amenaza de construcción aparece principalmente por comparar los criterios propuestos con cobertura de código. La validez de la cobertura de código como medida de efectividad de los conjuntos de test está aún siendo estudiada por la comunidad de testing. Sin embargo, para porder evaluar si las correlaciones obtenidas son razonables alguna medida de referencia es necesaria, y esta cobertura ha sido muy utilizada durante ya muchos años en la investigación en testing de software.

Por otro lado, los criterios propuestos son de caja negra y los resultados se están comparando con lo que en principio son considerados criterios de caja blanca. Esto podría ser cuestionable, ya que valores de referencia de un criterio de caja negra parecieran ser más adecuados. Sin embargo, no existe ningún estándar *de facto* entre los lenguajes de especificación de caja negra, y la elección de cualquiera de ellos podría estar introduciendo un sesgo en los resultados ya que es sabido que la estrucutra de una especificación puede tener un impacto importante en los criterios de cobertura [HGW04, RHOH98, PPW+05, RCW08, MBLDP11, Wei10]. La idea detrás de la elección del código es que éste constituye la especificación más detallada posible de su comportamiento, y que por lo tanto da una cota superior en cuánto a su calidad descriptiva del comportamiento del software subyacente.

Capítulo 10: Trabajo relacionado

Es este capítulo se compara el trabajo presentado en esta tesis con otros. Por un lado se referencian trabajos en los que se usan modelos en el contexto de testing de software, y por otro se hace hincapié en enfoques que abordan el problema de conformidad de protocolos.

Modelos para testing

La generación de tests para software con estado interno ha llamado la atención de muchos investigadores en los últimos años [Ton04, AML11, IX08, VPP06, LMS07]. Estos incluyen, entre otros, enfoques de random testing [PLEB07], enfoques evolutivos [Ton04], enfoques basados en cobertura de código [TXT+11] y enfoques basados en búsquedas [SGA+11].

Existen trabajos que explicitamente usan o hacen minería de modelos para generar tests de alta calidad. Por ejemplo, en [LMS07] se cocienta el espacio de estados de una clase utilizando los observadores booleanos. En [VPP06], los estados abstractos se computan usando abstraccion de forma, ignorando los valores concretos contenidos en los contenedores y tomando en cuenta únicamente la forma en que sus nodos están interconectados. En [DKM+10] se infiere y se usa un modelo de *typestate* (similar a los EPAs) para guiar la generacón de nuevos casos de test que tratan de cubrir las transiciones aún no cubiertas.

El trabajo presentado en esta tesis difiere con estos en diversas formas: (*i*) los enfoques mencionados no estudian especificamente el problema de conformidad de protocolos; (*ii*) no definen explícitamente criterios de adecuación de test; y (*iii*) no proveen evidencia estadística sobre la efectividad de cubrir las abstracciones usadas.

Testing de conformidad

Existe una gran cantidad de trabajos que se focalizan definir coberturas tanto a partir de especificaciones como de modelos usando muy diversos lenguajes [HBB+09]. Los enfoques existentes pueden clasificarse en dos categorías: criterios de cobertura

estructural (o basados en especificaciones) y criterios de cobertura semánticos. Los criterios de cobertura estructural se definen tanto en término de especificaciones [BPBM97, KKT07, OLAA03, Hie97, DF93, PRB09] o de código ejecutable generado a partir de especificaciones o modelos de simulación [PPW+05, RCW08]. A diferencia de lo que sucede con criterios de caja blanca, no existen demasiados trabajos que provean evidencia estadística para estos enfoques que son especialmente apropiados para testing de conformidad de protocolos.

Por otro lado, en enfoques semánticos, las coberturas son definidas en término de formalismos que directamente denotan el protocolo esperado de un software. Esta línea es la que se usó en trabajos fundacionales de testing de conformidad de protocolos usando máquinas de estado finitas [LY96]. Sin embargo, a diferencia del trabjo aquí presentado, esos trabajos asumen que espacios de estados finitos. Un trabajo que no asume la finitud de protocolos es [JT96], basado también en LTSs y usando IOCO como noción de conformidad. Sin embargo, usando este enfoque no se han definido medidas de cobertura.

Una forma de lidiar con modelos de comportamiento infinito es introduciendo abstracción. Existen diferentes técnicas de finitización: *unfolding* [BPBM97], *slicing* [VCG+08, GKSB11] y poda de estados [GKSB11]. Sin embargo, para ninguna de estas técnicas de finitización existen estudios estadísticos sobre la cobertura de las abstracciones generadas.

Resumiendo, el enfoque presentado en esta tesis entra dentro de la categoría de enfoques semánticos. El espacio de estados infinito se trabaja introduciendo abstracciones finitas que sobreaproximan el espacio de estados en forma similar a los *typestates* [BBA09] constituyendo un caso especial de partición de categorías, y se provee evidencia estadística sobre su efectividad.

Capítulo 11: Conclusiones y trabajo futuro

Esta tesis representa un paso al frente en la definición y el entendimiento de cómo coberturas semánticas sobre espacios de estados infinitos pueden ser usadas para el desarrollo de técnicas efectivas de testing de conformidad de protocolos. El problema planteado se ha abordado introduciendo abstracciones finitas. Un buen entendimiento de la relación entre detección de fallas, criterios de cobertura de caja blanca y cobertura del espacio de estados del espacio semántico derivado de especificacionese podría ayudar a mejorar random testing, selección de casos de test, técnicas de priorización de casos de test y, en general, el uso de heurísticas para la generación de casos de test a partir de especificaciones.

Los resultados experimentales son promisorios y sugieren que la cobertura de EPAs funciona bien como medida de calidad de conjuntos de tests en relación a detección de fallas de conformidad. Esto resulta particularmente importante en el contexto de testing de caja negra y constituye una oportunidad para la definición de criterios que sean independientes del lenguaje y las construcciones usadas en una especificación.

Los resultados también proveen evidencia de la correlación existente entre la cobertura de EPA y la cobertura de código. Esto puede tener implicancias prácticas en enfoques de desarrollo que pregonan que los tests sean escritos antes que las implementaciones. En estos contextos, desarrollar tests que busquen maximizar la cobertura de EPAs puede servir para generar tests con alta cobertura de código (que una vez que el código se encuentre disponible podrían ser extendidos de ser necesario).

El trabajo futuro estará centrado en estudiar otras abstracciones para la definición

de criterios de cobertura y comparar los resultados que se obtengan con los presentados en esta tesis. Por otro lado, también se pretende llevar a cabo experimentos con marcos más generales, que incluyan tanto APIs con comportamiento no determinístico como los valores de retorno en la definición de la relación de conformidad. También se pretende estudiar la relación de costo/beneficio que se produce al guiar la generación de casos de test mediante las coberturas propuestas.

Part I

Prelude

CHAPTER 1

Introduction

1.1 API Call Protocol Conformance Testing

Despite progress made, the automatic generation of efficient high quality test suites is still a major challenge for many kinds of software [FA12, XXTdH11, HBB+09]. This is the case for stateful components such as APIs, GUI, web software, protocol servers and clients that have non-trivial requirements with respect to the ordering in which their methods or procedures ought to be called to produce meaningful results or to access certain functionality [BKA11]. For instance, in Java the interface `ResultSet` is used to access or modify elements from a database. The call protocol for a `ResultSet` instance prohibits cursor movements while a new record is being inserted (i.e. no cursor movement between `moveToInsertRow` and `insertRow`) but allows closing the `ResultSet` even if insertion has not been completed. Some web-servers are another example of a stateful components: although they must be able to receive requests in any order, only by following a specific sequence of actions it is possible to trigger portions of its functionality. Components with non-trivial call protocols and options are particularly challenging for testing approaches [XXTdH11, GZE+12]. It is known that they are hard for both random testing and related approaches [GZE+12] and dynamic symbolic execution techniques [XXTdH11].

A particularly important type of testing for stateful components is done to gain confidence in that they conform to their intended *call protocols* (e.g., [GKSB11]). For instance, call protocol conformance is crucial to gain assurance that client code abiding to intended usage will not fail due to making calls on code that poorly implements the intended protocol (as the dual problem of typestate verification [DF04]). Thus, protocol conformance underlies settings like Model-Based Development [Sel03], Model-Based Testing [UL07] and Test-Driven Development [Bec03]. In this context, testing focus is on verifying that the code under test accepts or rejects sequences of method calls according to the intended call protocol. This is intractable as in principle it may consist in checking for a potentially infinite set of sequences of method calls that the implementation accepts or rejects, each one according to the call protocol.

White box adequacy criteria (i.e. adequacy criteria based on code) implies that the same test suite developed at design time may be adequate for some implementations but not for others. Such a situation is undesirable in, for instance, cross

platform development of API's which is becoming widespread in product family development. The question we set out to answer in this thesis is if it is possible to define adequacy criteria for testing call protocols that are independent of the structure of the code to be tested. That is, whether adequacy criteria can be defined in terms of the intended call protocol rather than its implementation.

Black box testing has addressed this problem to some extent. The vast majority of work on black box testing has studied structural strategies for defining adequacy of conformance with respect to specifications [HBB+09]. Coverage criteria are then defined either in terms of structural elements of the specification (e.g., [KKT07, OLAA03, Hie97, DF93, PRB09], etc.) or the executable code generated from it (e.g. [PPW+05, RCW08]). This yields criteria and empirical studies influenced by (accidental) elements of the structure of the model or its executable code such as predicates, control or data flow elements. Authors have already warned that accidental aspects of specifications or model compilers may potentially influence effectiveness of criteria (e.g., [PPW+05, RCW08, HGW04, RHOH98, MBLDP11, Wei10], etc.). Moroever, being tightly coupled to a particular language, the relation between the criterion and protocol state space actually covered and, consequently, the degree to which the failure domain is explored, is not studied. This in turn hinders the generalisation of the empirical results of this body of work [HBB+09] to the general problem of black box testing of call protocols.

1.2 Towards Semantic Coverage Criteria

Although some sort of semantic coverage would be expected as a natural measure of testing, there is a hitherto unexplored difficulty when the behaviour of the system under test is infinite. One of the main challenges of defining effective semantic or behaviour-based coverage criteria is that call protocols are typically infinite state and consequently traditional black box criteria for conformance testing of call protocols are not applicable as finite state spaces are assumed [KKT07, OLAA03, MBLDP11, DASC93, MCLH05, CYH06, Hie04]. Various strategies for finitising protocol state spaces have been studied [DKM+10, LMS07, VPP06]. However, there are no empirical results on their effectiveness.

> *Our general hypothesis is that effective notions of behavior coverage are actually feasible by defining them in terms of finite abstractions defined over the semantic domain that describes the intended call protocol behaviour*

In this thesis we propose different coverage criteria over a finite abstraction of infinite state behaviour call protocols and present experiments in which results show that the criteria are good predictors of fault detection in the context of conformance testing. The practical implications of this result may be that in the context of development approaches which advocate test development before coding, generating tests according to an abstraction of the call protocol semantics of an artefact with non-trivial requirements on method call ordering would provide a good criterion for detecting faults.

The coverage criteria are defined over enabledness preserving abstractions (EPAs) [dCBGU11]. These abstractions quotient an infinite state space into finite classes of states which allow the same method calls. They also abstract method parameters through existential elimination. The transitions between EPA states (corresponding to action invocations that may lead the program from one state to another) are the

key of the first proposed criteria: the more transitions are covered by a test suite, the higher the level of adequacy according to the criteria.

1.3 Experimental Evaluation

We evaluate fault detection ability of EPA transition coverage on five industrially relevant Java classes with rich call protocols by analysing the mutant detection capability (i.e., detection of mutants that throw unexpected exceptions or timeout) of randomly generated test suites. The experimentation was designed to answer a number of research questions (RQs). Succinctly, we study (*RQ.1*) how good EPA transition coverage is for fault detection by looking at (*RQ.1.1*) EPA Transition Coverage and fault detection correlation, and (*RQ.1.2*), as achieving higher coverage is related to test suite size, the impact of EPA transition coverage over fixed-size test suites. We then perform a more qualitative study (*RQ.2*) aiming to provide insight on why the positive results for RQ.1 are obtained. We look at correlation between EPA transition coverage and code coverage (*RQ.2.1*), if the category partition implicitly defined by EPA transition coverage (an input i belongs to the subdomain defined by a transition t if and only if i exercises t) is likely to have dense subdomains (*RQ.2.2*) and the effectiveness of individual EPA transitions (*RQ.2.3*). We also investigate (*RQ.3*) if the results for RQ.1 and RQ.2 are not simply an artefact of achieving action coverage. We compare the correlation between action coverage and fault detection with those obtained for EPA transition coverage (*RQ.3.1*), the failure rates of the subdomains derived from both actions and transitions (*RQ.3.2*), and the effectiveness of actions and transitions for revealing faults (*RQ.3.3*). Derived from resutls of RQ.3, we propose and study (*RQ.4*) a combined criterion that involves covering EPA transitions and actions. We investingate the correlation of the combined criterion for fault detection (*RQ.4.1*), and also perform the analysis for fixed-size test suites (*RQ.4.2*). Finally, we investigate the quality of EPA-coverage-based proritisation techniques and compare them to techniques based on code coverage and random orders (*RQ.5*). We examine the effects of using random, EPA and code coverage criteria on the fault detection rates of the resulting prioritised test suites, by comparing the APFD values [EMR00] obtained in each case.

Subject to the threats derived from the adopted failure model (which are further discussed in Chapter 9), results suggest that the EPA Transition Criterion is a good predictor of mutant detection (RQ.1.1). Results also show that EPA transition correlations are comparable or better than those of structural (rather than behavioural) white box criteria. They also suggest that the EPA Transition Coverage Criterion, which is behavioral and hence independent of specification language bias, performs comparably in terms of predictability of test suite fault detection, resulting in higher values for all case studies against statement coverage and is better than branch coverage in two out of five case studies. Moreover, for fixed-size, test suites with the highest behavioral adequacy are statistically better (RQ.1.2) in terms of fault finding.

Results for RQ.2 aiming at getting a deeper understanding of the observed phenomena reinforce the results of RQ.1. Results indicate that the proposed criterion is a good predictor of structural coverage criteria (statement and branch coverage) when applied over code under test (RQ.2.1). These results indicate that a first and early shot at producing high code coverage test suites (which could be extended when code is available) can be achieved through EPA transition coverage.

Results also suggest that the domain partition implicitly derived from EPA transitions is likely to produce subdomains that are dense in failures, i.e., a subdomain

with high failure rate (RQ.2.2). This is close to what is known to be optimal in the context of partition testing [JW89].

The finer grained study of RQ.2.3 investigates the effectiveness of transitions (the likelihood of revealing a mutant when traversing the transition rather than by the subdomains defined by them) reveals that for each fault there is almost always one transition that is highly effective in detecting it ($> 90\%$ effectiveness) while nearly all the rest of the transitions have poor effectiveness ($< 10\%$). Furthermore, we show that the effective transition is not always the same one, it depends on the fault. This is in line with previous results: as faults are a-priori unknown, it makes sense to consider more adequate those tests that cover more transitions. Test suites with highest adequacy should cover all transitions, and consequently the one highly effective is exercised. However, a more significant implication is that, as uniqueness of effective subdomain per fault does not hold but does for transitions, there is an opportunity to improve on EPA transition coverage criterion.

Experimentation regarding RQ.3 aimed at investigating that promising results for EPA transition coverage are not simply because the criterion is a refinement of an action coverage criterion. Results of RQ.3.1 suggest that for correlation with fault detection and structural coverage, in most cases EPA transition coverage performs comparably or better than action coverage. Results of RQ.3.2 indicate that subdomains defined by actions are less dense than those derived from transitions. Finally, results of RQ.3.3 show that transitions are, by far, much more effective than actions for exposing faults. These results suggest that EPA transition coverage can provide benefits over action coverage, and justify studying a combined criterion.

The set of results corresponding to RQ.4 shows that considering both actions and transtions for measuring the adequacy level of test suites yields to results that outperform those of actions and transitions (when considered in isolation) as a fault detection predictor (RQ.4.1). Also, they show that for fixed-size test suite, those with higher adequacy level according to the combined criterion are better as regards fault detection (RQ.4.2).

Finally, regarding RQ.5 we have found that priritisation based on EPA-coverage is always better than a random approach, and also that they tend to achive high failure-rates almost at the same speed or faster than those based on white-box code coverage.

In conclusion, we believe that this thesis is a step to understanding how behavioral coverage of call protocol behaviour correlates with fault detection in the context of protocol conformance testing. Results suggest, to the extent of the external validity threats of our experiments, that the proposed criteria are good predictors for fault detection and for classical structural coverage criteria such as statement and branch coverage. Besides, transitions have a very interesting property as regards effectiveness (much better than the effectiveness of actions), and we define another criterion that is even better which take advantage of this phenomena. The implication being that the criterion could help improve random testing, test driven development, test case selection and, in general, techniques for tests generation from formal specifications when applied to API code with rich call protocols. The results seem to indicate that such approaches would benefit from introducing heuristics that aim to maximise action and EPA coverage.

1.4 Contributions

The contributons of this work can be summarized as follows:

1. the idea of using behaviour abstractions in the context of call protocol conformance testing;
2. the definition of testing adequacy criteria based on enabledness-preserving abstractions (EPAs);
3. a thorough experimental evaluation of the proposed criteria, which includes:
   (a) a study of the quality of the proposed criteria as test suite fault predictors;
   (b) a study of the quality of the proposed criteria as code coverage predictors;
   (c) a study of the results from a categroy partition point of view;
   (d) a fine-grained study of the realation between transitions and fault detection;
   (e) the study of the differences between the effects of covering actions and covering transitions;
   (f) a comparison of the proposed criteria against white-box based and random techniques in the context of test suite prioritisation.

1.5 Roadmap

Part I concludes in chapter 1, and the rest of the document is structured as described below.

Part II is devoted to formally introduce the addressed problem and to describe the design of the experiment conducted. In Chapther 2 we formalise the problem by defining what is meant by an intended call protocol to be provided by a code artefact, the actual call protocol implemented by a code artefact, and the conformance relation that is expected to hold between them. We also define the EPA transition test adequacy criterion and then formulate the research questions that we address in Part III. In Chapter 3 we describe the design of the experiment, the subjects on which the experiment was carried on, how their EPAs are constructed and the fault model adopted.

Part III is the core of the thesis, where all the research questions are developed and analysed. Chapter 4 addresses RQ.1 and seeks to quantitatively analyse the fault detection ability of the coverage criterion defined in Chapter 2. Chapter 5 deals with RQ2., which aims at providing more qualitative results that may provide insight on why positive results for RQ.1 are obtained. Chapter 6 tackles RQ3, which aims at investigating if the results for the proposed criterion are not simply because the criterion is a refinement of an action coverage criterion. Chapter 7 is concerned to RQ4. There we define and investigate a coverage criterion that combines actions and EPA transitions, and we show that there are some benefits in this combined criterion. Finally, in Chapter 8 we address RQ.5 by analysing the proposed criteria in the context of test case prioritisation.

We close this work in Part IV. In Chapter 9 we present and discuss the threats to validity regarding the results presented in previous chapters. We analyse the related work in Chpater 10 and conclude in Chapter 11 with some final words and an overview of future lines of research derived from this work.

Part II

Problem Statement & Experiment Design

CHAPTER 2

Problem Statement

We are interested in studying behavioral coverage criteria for testing protocol conformance. In this chapter we formalise the problem by defining what is meant by an intended protocol to be provided by a code artefact, the actual protocol implemented by a code artefact, and a conformance relation that is expected to hold between them. We also define the EPA transition test adequacy criterion and then formulate the research questions that we address in Part III.

2.1 Protocol Conformance

The term *protocol conformance* has been extensively used, encompassing many different approaches related to checking if an implementation is in compliance with a specification. In this work we use a more specific term: API call protocols. As in the case of typestate verification literature [BKA11, FGRY05, FYD+08, BA07, BBA09] the ultimate verification purpose here is the very basic interoperability preservation: to be conformant, an API implementation must support sequences of calls that a given specification (potentially used to build or verify client code) defines as legal. As usual in this context, our setting is restricted to determinstic specifications and implementations with output abstraction. Determinism in the specifications is a natural assumption that provides API clients with certainties about when it is legal to call a given operation. On the other hand, assuming (failure) determinism for the API implementations is a reasonable asumption in some contexts that greatly simplifies presentation and experimentation as further explained in the sequel.

Full formal treatment of programming language semantics is beyond the scope of this thesis. We provide an intuitive definition, sufficient for defining rigorously the notion of protocol conformance that we use. The semantics of an API implementation can be defined as a call protocol labelled transition system (LTS). The states of the LTS are all configurations of the internal state of the code. If the code is a class, then configurations correspond to all structurally distinct instances of the class. If the code is an API implementation, configurations are all possible valuations on internal variables of the API. Transitions are the effect of successful invocations of specific methods with concrete parameters. A transition between states s and s' will be present if and only if the execution of the associated method -with the annotated actual parameters- on the configuration corresponding to s eventually halts, does not yield any exception and changes the internal state of the code to a configuration

that corresponds to s'.

A specification language designed to describe the intended call protocol behaviour of a class or API to be developed can be given semantics in a similar fashion. The *intended call protocol LTS* defines which are the (potentially infinite) set of valid method invocation sequences on a code artefact (each invocation including actual parameters). An implementation is conformant if it accepts the sequences of method invocations that are legal according to the intended protocol.

Hence, in this work we adopt (intended and actual) call Protocol LTS as the semantic domain for implementations and specifications. The actual protocol LTS represents the real behaviour of the implementation while the intended protocol LTS represents the intended behaviour according to some specification. Both LTS are semantic representations and independent of the programming and specification languages used.

Definition 1 (Call Protocol LTS)**.** *Let $m_1, \ldots, m_n$ be method names, and $\mathcal{D}_i$ the domain of m_i. A LTS protocol for $m_1, \ldots, m_n$ is a tuple $L = \langle \Sigma, \mathbb{S}, \mathbb{S}_0, \Delta \rangle$ where,*

- *$\mathbb{S}$ is the (possibly infinite) set of states.*
- *$\mathbb{S}_0$ is the initial state.*
- *$\Sigma = \bigcup_{i \leq n}(\{m_i\} \times \mathcal{D}_i)$ is the set of possible method invocations.*
- *$\Delta : (\mathbb{S} \times \Sigma \times \mathbb{S})$ is the transition relation that maps pairs of a state and method invocation to the corresponding resulting state. The relation must be a partial function on the first two elements of the tuple.*

As said, we do not make assumptions on the way the *intended protocol LTS* of a code artefact is described: we simply assume that the semantics of such language can be defined in terms of a call protocol LTS as defined above. In fact, there are several ways an *intended protocol LTS* can be defined in practice: it could be formally given as a model in a MBT setting, it could be defined by a reference implementation, it could be given as a set of known valid traces, it could be mined from client applications of the code, etc. In any case, an LTS is a reasonable representation of the underlying behavior. Note that we require protocol LTS to be deterministic.

The expresion $s \rightarrow^{m_i(p)} s'$ denotes that $(s, m_i(p), s') \in \Delta$, $s \rightarrow^{m_i(p)}$ denotes $\exists s'.(s, m_i(p), s') \in \Delta$, and $s \not\rightarrow^{m_i(p)}$ denotes $\nexists s'.(s, m_i(p), s') \in \Delta$. These definitions are trivially extended to sequences of method invocations. Conformance between call protocol LTS is defined as an inclusion with respect to the sequences of method invocations they accept.

Definition 2 (Call Protocol LTS Conformance)**.** *Given an intended and an actual Call Protocol LTS, I and A, over the same set of methods with initial states $\mathbb{S}_{\mathbb{I}0}$ and $\mathbb{S}_{\mathbb{A}0}$, we say that A is in conformance to I if and only if for all sequence w of method invocations -with concrete parameters-, $\mathbb{S}_{\mathbb{I}0} \rightarrow_I^w$ then $\mathbb{S}_{\mathbb{A}0} \rightarrow_A^w$.*

The *failure model* we adopt in this thesis is derived from the definitions of intended and actual protocol LTS and the definition of conformance. A *failure* is then a sequence of method invocations with concrete parameters which is part of the intended protocol but does not terminate or raises an exception when executed on the implementation.

Definition 3 (Failure)**.** *Given an intended Protocol LTS I and an actual Protocol LTS A over the same set of methods with initial states $\mathbb{S}_{\mathbb{I}0}$ and $\mathbb{S}_{\mathbb{A}0}$, we say that a sequence w of method invocations -with concrete parameters- is a failure if $\mathbb{S}_{\mathbb{I}0} \rightarrow_I^w$ and $\mathbb{S}_{\mathbb{A}0} \not\rightarrow_A^w$.*

Figure 2.1 shows a portion of the intended call protocol LTS of a bounded Stack over alphabet $\{a, b\}$. Initially the stack is empty, and thus the only possible action in S_0 is to push an element into it. Note that there are two outgoing transitions from S_0 labelled with action *push*, but each of them corresponds to a different actual parameter. Let us consider now the faulty Java implementation of a bounded Stack shown in listing 2.1. It fails when trying to pop the last element of the stack: the `IllegalStateException` should be thrown when `top` is less than 0, but not when it equals 0. Figure 2.2 shows part of its actual protocol LTS. As above mentioned, it fails when `pop` is invoked on a `BoundedStack` containig only one element. Consequently, there are no transitions to go back to the initial state. Therefore, the sequence $[push(a), push(a), pop(), pop()]$ is a failure, since it is part of the intended call protocol but not of the actual call protocol. In practice, a client following the intended call protocol may fail if the implementation is non-conformant.

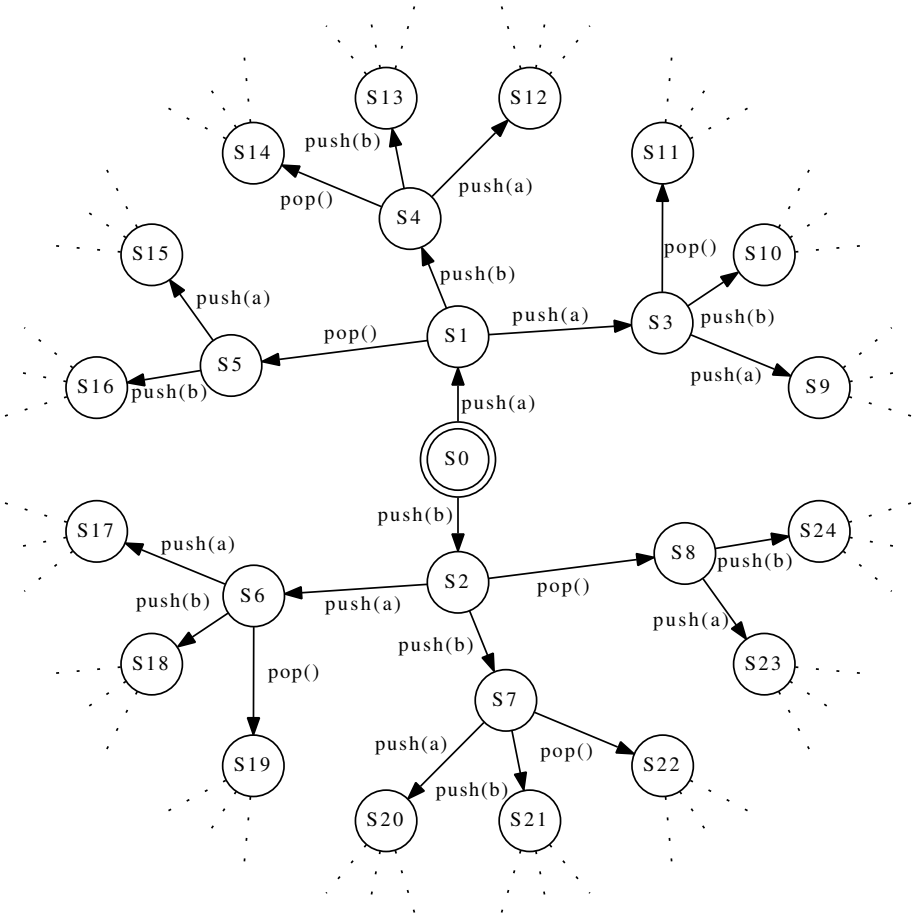


Figure 2.1: Snippet of the intended protocol of a bounded stack

2.2 Enabledness Preserving Abstractions (EPAs)

We aim to define an adequacy criterion based on its protocol's behaviour. Therefore, the criterion should be independent of the specification language used to express the intended protocol and the programming language used to implement it.

```
public class BoundedStack {
    Object[] elems;
    int top;

    public void BoundedStack() {
      elems = new Object[1000];
      top = -1;
    }

    public Object pop() {
        if (top <= 0) {
          throw new IllegalStateException();
        } else {
          return elems[top--];
        }
    }

    public void push(Object o) {
      if (top >= elems.length-1) {
          throw new IllegalStateException();
      } else {
          elems[++top] = o;
      }

    }
}
```

Listing 2.1: Faulty implementation of a bounded stack

Classical protocol testing approaches are defined over finite state machines [LY96], however protocol behaviour is typically infinite state. We propose to work over a *finite state abstraction* of the intended protocol. The abstraction we propose is the enabledness preserving abstraction (EPA) from [dCBGU11]. Basically, an EPA is a LTS where labels are method names (with no concrete parameters). EPAs abstract the state space of the protocol LTS by quotienting it according to the methods that are enabled. In other words, two states of a protocol LTS are represented by the same abstract state if and only if for every method and concrete parameters enabled in one state, the same method is enabled in the other state (possibly with different concrete parameters). Consequently, an EPA state can be thought of as representing a particular subset of methods and abstracting all concrete states for which for every method in the subset there is some concrete parameters for which calling that method on that state is valid in the protocol LTS. EPAs abstract parameters by introducing a transition between abstract states only if there exist parameter values such that a concrete state of the source abstract state can lead to a concrete state of the target abstract state (i.e. existential elimination). Figure 2.3 shows the EPA of the Intended Protocol LTS of a bounded stack depicted in Figure 2.1 [1]. State S_0 abstracts concrete states in which the only enabled action is *push* (an empty stack); S_1 abstracts states where *push* and *pop* are both enabled (a neither full nor empty stack); and S_2 abstracts concrete states where the only enabled action is *pop* (a full stack).

We define EPAs formally by first defining what is meant for two states of a Protocol LTS to be enabledness equivalent.

[1] Note that the single arrow from S_1 to S_1 represents two different transitions, one for *push* and other for *pop*.

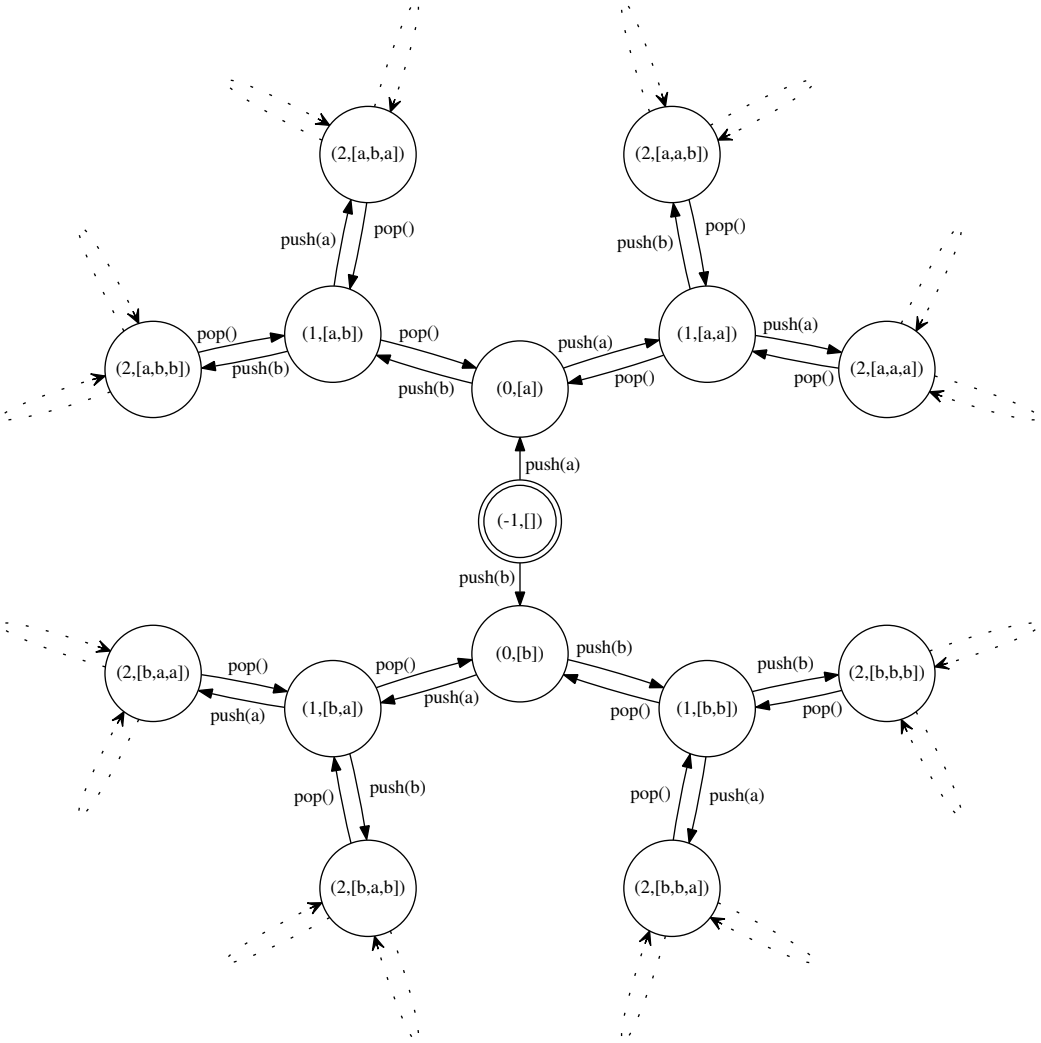


Figure 2.2: Actual protocol of the faulty implementation of the bounded stack

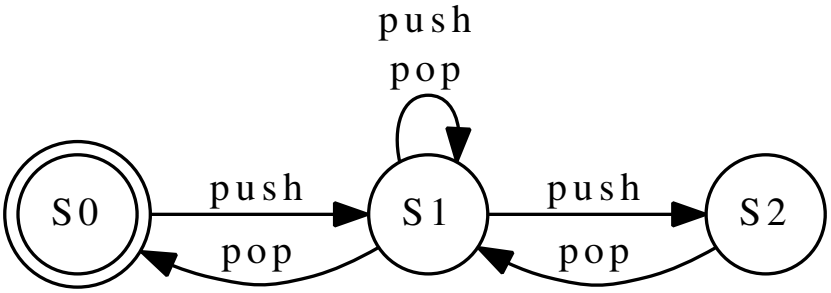


Figure 2.3: EPA of a bounded stack

Definition 4 (Enabledness Equivalent States)**.** Given a protocol LTS $L = \left\langle \Sigma = \bigcup_{i \leq n}(\{m_i\} \times \mathcal{D}_i),\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \right\rangle$ over method names $m_1, \ldots, m_n$ and $\mathcal{D}_i$ as the domain of m_i, and two states $s_1, s_2 \in \mathbb{S}$, we say that s_1 and s_2 are *enabledness equivalent* states (noted $s_1 \equiv s_2$) if and only if for every m_i $\exists p \in D_i.s_1 \rightarrow^{m_i(p)} \iff \exists p' \in D_i.s_2 \rightarrow^{m_i(p')}$.

States S_1 and S_2 of the intended protocol of Figure 2.1 are enabledness equivalent, since both states enable the same set of actions (note that S_4, S_6 and many

```
public void test() {
  BoundedStack bs = new BoundedStack();
  bs.push('a');
  bs.pop();
  assertEquals(0,size(bs));
}
```

Listing 2.2: Sample unit test for a `BoundedStack` object

other states are equivalent to S_1 and S_2 as well). Given an LTS describing a protocol, we now define its enabledness-preserving abstraction as a finite (potentially non-deterministic) state machine which groups the protocol states according to the actions that they enable.

Definition 5 (Enabledness-preserving Abstraction)**.** Given a protocol LTS $L = \left\langle \Sigma = \bigcup_{i \leq n}(\{m_i\} \times \mathcal{D}_i),\ \mathbb{S},\ \mathbb{S}_0,\ \Delta \right\rangle$ for a protocol over method names $m_1, \ldots, m_n$ and $\mathcal{D}_i$ as the domain of m_i, we say that the LTS $M = \langle \bigcup_{i \leq n}(\{m_i\}), S, S_0, \delta \rangle$ is the *enabledness-preserving abstraction* (EPA) of L if there exists a total function $\alpha : \mathbb{S} \to S$ such that $\alpha(\mathbb{S}_0) = S_0$ and for every $s, s' \in \mathbb{S}$ and every method name m_i, $(\alpha(s), m_i, \alpha(s')) \in \delta \iff \exists p \in D_i . s \to^{m_i(p)} s'$. Furthermore, given a pair of states s_1, s_2 on $\mathbb{S}$, it holds that $s_1 \equiv s_2 \iff \alpha(s_1) = \alpha(s_2)$.

EPAs were used in previous work for validation of specifications [dCBGU09] and programs [dCBGU11]. Also, there is tool support for constructing EPAs either from a contract-based specification [dCBGU09] or from source code [dCBGU11].

2.3 EPA Transition Coverage Criterion

As mentioned in [MV95], naively meassuring coverage of a test suite on an infinite state space would typically yield zero as result. This would be the case if we try to meassure coverage on the entire protocol LTSs. Considering that they can be abstracted as EPA models, it is natural to define and investigate whether the latter would be effective as a basis for defining a coverage criterion for conformance testing. In this thesis, we define one criterion based on transition coverage and we study its effectiveness as test suite quality predictor.

Let us begin by defining a *unit test* as a sequence of method invocations with concrete parameters and a *test suite* as set of unit tests. Note that the effect of the execution of a unit test over an instance can be univocally interpreted as a path along a protocol LTS. In turn, (and because EPAs can simulate all paths of the LTS they abstract and each protocol LTS state is abstracted by only one EPA state) a path on the protocol LTS can be unequivocally simulated by a path in the EPA by applying the abstraction function α. We call the later an *α-abstracted execution* of a unit test. Listing 2.2 shows an example of a unit test for a `BoundedStack` object and Figure 2.4 its corresponding α-abstracted execution. Note that the non-conformant implementation of Listing 2.1 throws an exception every time the α-abstracted execution of a unit test traverses the transition (S_1, pop, S_0).

We now define the adequacy criterion over EPAs of protocol LTS, which simply consists on covering the transitions of the protocol's EPA.

Definition 6 (EPA Transition Adequacy Level)**.** *Let L be a protocol LTS, A its EPA and TS a test suite. The adequacy level of TS with respect to A according to the EPA Transition Coverage Criteria is defined as the percentage of transitions of A that are covered by the α-abstracted execution of the unit tests of TS.*

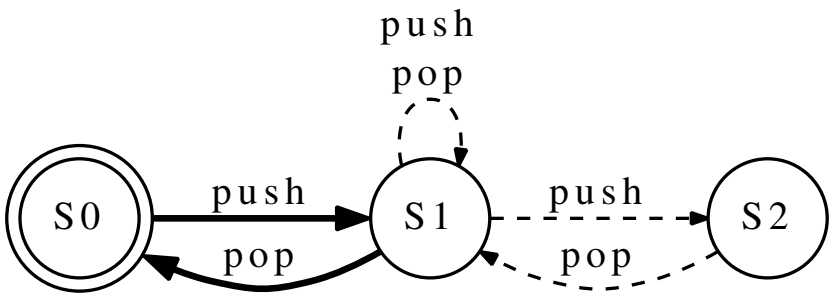


Figure 2.4: α-abstracted execution of the unit test over the EPA of the intended Protocol LTS of a bounded stack

is defined as the percentage of transitions of A that are covered by the α-abstracted execution of the unit tests of TS.

As an example, the adequacy level of a test suite consisting only of the unit test of Listing 2.2 is 33, since it covers two out of six transitions. In particular, `push` is only invoked on an empty stack. A test suite with adequacy level 100 should necessarily contain in addition a unit test in which `push` is executed on a non-empty stack leading to a full stack and one unit test in which `push` is executed on a non-empty stack leading to a stack that is neither empty nor full.

2.4 Research Questions

We now pose the research questions that are the focus of the experimentation reported in the Part III of this thesis. The first question aims at establishing if the EPA Transition Adequacy Level is a good predictor of the ability of a test suite to detect faults that manifest themselves as violations of the intended protocol, i.e. protocol conformance failures.

RQ.1. To what extent does the EPA Transition Adequacy Level predict the ability of test suites to detect faults in the context of call protocol conformance testing?

We first study this question by looking at the correlation between the coverage of EPA transitions and the ability of a test suite to detect faults.

RQ.1.1. Does the EPA Transition Adequacy Level have a high correlation with fault detection in the context of protocol conformance testing?

Correlations are typically considered high when equal to or above 0.7. However, we believe that a comparative measure would provide a more robust answer to RQ.1.1. Consequently, we also look at how well branch and statement coverage performed over the code of the subjects (i.e., as white box criteria) would predict test suite fault detection ability. The importance of this analysis is twofold. First it provides a baseline reference for correlations yielded by the proposed criterion. Second it enables a comparison against what would be measuring structural adequacy using an ideal specification in terms of closely mimicking the structure of code

under test. Choosing the code of the reference implementation as the specification language against which to compare EPA coverage is discussed in Chapter 9.

Since requiring coverage leads naturally to requiring larger test suites, it is standard to analyse if stronger correlations are just a consequence of size or if the criterion has an independent effect on test suite quality. More specifically, we aim to study if picking a test suite with higher adequacy is more likely to detect more faults than picking a test suite of the same size but with lower adequacy.

RQ.1.2. Given a fixed test suite size, do test suites with higher EPA Transition Adequacy Level perform better in terms of fault detection than those with lower level of adequacy?

RQ.2 aims at providing some insights that can explain the results of RQ.1.

RQ.2. Why does EPA Transition Adequacy Level predict the ability of test suites to detect faults that manifest themselves as call protocol conformance failures?

Although code coverage do not always correlate to test suite effectiveness [IH14], programs are expected to be tested using test suites that achive high structural coverage (otherwise portions of the program are under-tested). Therefore, achiving high code coverage can be considered a necessary (but not sufficient) condition for test suites. Hence, a high correlation between EPA transition coverage and code coverage could partially explain the results of RQ.1.

RQ.2.1. Does EPA Transition Adequacy Level predict the statement and branch coverage?

For RQ.2.1. we analyse the coverage of each test suite over a reference implementation and compute correlations. We also, as before, analyse the impact of EPA coverage on code coverage for fixed size test suites to account for dependencies between coverage and size.

Analytical results in [JW89] indicate that category partition approaches that define dense subdomains (i.e. subdomains that have a high likelihood of identifying a fault) are likely to be better at identifying faults. Any coverage criterion implicitly defines a category partition, in the case of EPA transition coverage, a subdomain for a transition can be defined as all the tests that when executed cover the transition. If such implicit category partition indeed defines dense subdomains, this would contribute to explaining results in RQ.1.

RQ.2.2. Does the category partition implicitly defined by EPA transition coverage result in subdomains with high failure rates?

Although a test in the subdomain defined by a transition covers that transition, it may reveal a fault at any point of the test. Thus, RQ.2.2 provides a rather coarse

grained insight on the effectiveness of each transition in detecting faults. A finer grained question would be to investigate the likelihood that each transition has, when exercised, of revealing a fault. If every fault has at least one transition that is highly effective in detecting it, then it is reasonable to think that the higher the coverage the more likely the effective transition is covered.

RQ.2.3. Do faults have transitions that are highly effective in detecting non-conformant implementations?

The third main question we investigate relates to differentiating the effect of action coverage and transition coverage. Each transition of an EPA is associated with a particular action in a many-to-one relation. For instance, in the EPA of a bounded Stack shown in Figure 2.3 there are three different transitions corresponding to action `push`, and three different transitions corresponding to action `pop`. Clearly, achieving high EPA transition adequacy requires a degree of action coverage. Thus, we want to determine whether the observed phenomena are really determined by transition coverage. In RQ.3 we revisit previous questions for action coverage and compare against results obtained for EPA transition coverage.

RQ.3. How do results of previous questions compare when Action Coverage is considered instead of EPA Transition Coverage?

For addressing RQ.3, we first analyse the action coverage criterion as fault detection and structural coverage predictor. We do this by calculating correlation values for the Action criterion and compare them to those obtained for EPA Transition coverage.

RQ.3.1 How do Action Coverage Criterion correlation values for fault detection and structural coverage compare to those of EPA Transition Coverage?

Then we compare both criteria from a category partition perspective. That is, we compare the failure rates of the subdomains implicitly derived from Action Coverage and EPA Transition coverage.

RQ.3.2 How do failure rates of subdomains defined by Action Coverage compare to those of subdomains defined by Transition Coverage?

Finally, we study the likelihood that each action has, when executed, of revealing a fault. Again, we compare the results to the effectiveness of transition for exposing faults.

RQ.3.3 How do effectiveness of actions compare to the effectiveness of transitions for revealing faults?

Derived from results of previous main research questions, in the fourth one we investigate if action coverage and EPA transition coverage can be combined to get a better criterion. That is, a coverage criterion that exploits the benefits of covering both, actions and transitions. Again, we revisit RQ1. and RQ2. to evaluate the combined criterion.

RQ.4. How do results for RQ.1 and RQ.2 compare when a combined Action-EPA transition coverage criterion is adopted?

First, we study this question by looking at the correlation between the coverage according to the combined criterion and the ability of a test suite to detect faults and achieve structural coverage, and compare the results to those of Action coverage and EPA Transition coverage.

RQ.4.1 How do the combined criterion correlation values for fault detection and structural coverage compare to those of EPA Transition Coverage and Action Coverage?

As before, we also study the combined criterion in a "by size" basis.

RQ.4.2. Given a fixed test suite size, do test suites with higher Adequacy Level according to the combined criterion perform better in terms of fault detection and structural coverage than those with lower adequacy level?

Finally, the last research question focuses on the quality of EPA-coverage-based criteria in the context of test suite prioritisation. We investigate the fault detection rates of prioritised test suites acording to techniques based on these criteria, and compare the results with those of techniques based on code coverage and random orders.

RQ.5. How do EPA-based coverage prioritisation techniques compare to random and code coverage ones in terms of fault detection rate?

CHAPTER 3

Experiment Design

3.1 Experiment Overview

The aim of our experiments is to measure the predictive ability of the proposed adequacy criterion for *fault detection*. That is, to measure the strength of the correlation between this variable and the extent to which a test suite covers an EPA. The number of faults detected is the number of discrepancies between the intended and the actual protocol revealed by a test suite.

To answer the research questions proposed in Section 2.4 various types of values associated to test suites must be collected: *faults detected*, *code coverage*, *EPA transition coverage*, and *action coverage*. Our experimental strategy is based on code mutations as in [ABLN06, AZ03, BLW04].

Fault detection involves (*i*) fixing both an intended protocol LTS and a conformant implementation and (*ii*) obtaining implementations that fail to conform to the intended protocol in diverse ways. In general, this would imply getting a specification of the intended protocol together with a conformant implementation. That may introduce a major threat to validity as there is no guarantee that an implementation is conformant respect to a specification. For avoiding such risk, our strategy involves selecting an implementation as a *reference implementation* to be used as both the specification and implementation of the intended behaviour. This way, the actual protocol LTS of the subject implementation is conformant by construction. Also note that the *intended protocol LTS* is in fact the actual protocol LTS of the reference implementation. This may be a typical case for regression testing when the actual protocol of the original version is intended to be preserved by the newer versions.

We also use the reference implementation as the basis for generating non conformant or *faulty implementations*. We obtain faulty implementations by applying mutation operators to the reference implementation. Identification of failures is done by executing unit tests on both the reference implementation and on the mutated implementations. When mutation is unable to execute a sequence of calls that is valid in the reference implementation then the mutant is considered to be killed. More precisely, a failure is recorded if the mutated version throws an exception or timeouts on method call sequences that neither raise exceptions nor timeout in the reference implementation. Or in other words, if a unit test is a witness to a conformance failure between the intended protocol LTS and the actual protocol LTS of

the mutated implementation.

Note that semantically different mutations do not necessarily alter the actual protocol. For instance, altering the way an index is updated may (or may not) eventually lead to a state where some operation yields an exception. Between 56% and 70% of mutations (depending on the case study) produce faulty implementations. To have a representative set of flawed implementations, we decided not to filter a priori the applied mutation operators. As in [ABLN06], mutants that were not killed by any unit test (i.e., no test sequence led to an unexpected behaviour of that mutant) were considered mutations that have the same actual protocol as the reference implementation of the class.

3.2 Subjects

We restricted the universe of potential subjects to one programming language to allow for a uniform experimental platform regarding mutation, test generation and infrastructure for detecting failures. In particular, we fixed the language to Java to take advantage of existing tools and the availability of Java classes that satisfy our general criteria for subject selection: (*i*) code that features a rich set of restrictions on the order in which methods should be called (i.e., rich call protocols); (*ii*) code that is of industrial relevance; and (*iii*) code for which its EPAs can be obtained (see Section 3.3).

We performed studies on 5 subject classes: `Signature`, `ListIterator` and `Socket` from the Java Development Kit (JDK) 1.4 implementation; the `SMTPProcessor` class of JES mail server, a Java SMTP and POP3 e-mail server; and `JDBCResultSet` class, which is the implementation of the `ResultSet` interface of the JDBC specification of HyperSQL 2.0.0 database.

The Java `Signature` class is used to provide applications with the functionality of a digital signature algorithm. There are three phases to the use of a `Signature` object for either signing data or verifying a signature: (*i*) Initialization, with either a public key, which initializes for verification, or a private key, which initializes for signing; (*ii*) Updating, which updates the bytes to be signed or verified; and (*iii*) Signing or verifying a signature on all updated bytes. The `ListIterator` class provides functionality to go through the elements stored in a list. If the end of the list has not been reached, the iterator can retrieve the next element. Conversely, it can retrieve the previous element if the current index is not 0. It has methods for adding, removing and replacing elements in the list. The `Socket` class provides the client-side functionality to establish a TCP connection between two hosts. A `Socket` instance can open and close connections to a server and also can operate on streams for either send or receive data, among other operations. Note that we use a smaller version of the class in which a subset of the public methods are considered. Methods considered are those that are part of rich protocol requirements for the class. This version was previouly used in [CBGU13] and [HJM05]. The `SMTPProcessor`, which is a core class of the Java Email Server, is responsible for processing all incoming SMTP requests. It checks whether a request is valid or not in terms of the restrictions defined on the SMTP protocol specification. In presence of valid requests it process them, whereas it rejects the invalid ones. Finally, the `JDBCResultSet` class implements the `ResultSet` interface of the JDBC specification. A `ResultSet` represents a set of data which is generated by executing a query to a database. The class allows iterating over the result and making updates on the underlying database. It maintains a cursor pointing to the current row of data and provides methods to scroll the data back and forth. Also, it supplies methods for

making updates on the underlying database.

3.3 Construction of EPAs of Intended Protocol LTSs

The enabledness preserving abstractions of the intended protocol for each subject are built from the original reference implementations which act as infinite state specifications. We used different construction strategies for each case study.

A key resource for all subjects was the tool CONTRACTOR [dCBGU09]. It constructs EPA either from contract-based specifications [dCBGU09] or directly from source code [dCBGU11]. CONTRACTOR's output models over-approximate EPAs in the sense that they may include spurious transitions. When computing the model from contracts, the tool relies on a first-order theorem prover. If neither a proof nor a counterexample can be found to determine whether a transition exists or not, the tool simply adds it because it may be present in the EPA. In a similar way, when the calculation is made from code it uses a software model checker that tries to determine whether some statements present in the code are reachable or not (which is undecidible in the general case). Again, when no answer is found, CONTRACTOR conservatively adds the corresponding transition (see [dCBGU09, dCBGU11] for details).

Since we require intended protocol to be the actual protocol of the reference implementation, the EPAs of the intended protocol for subjects `Signature`, `ListIterator`, `Socket` and `SMTPProcessor` were obtained by using CONTRACTOR on code of the reference implementation as in [dCBGU11]. In all cases, no spurious transitions were generated, so we can be sure that the constructed EPAs are those of the actual protocols of the reference implementations. On the other hand, the EPA of the actual protocol LTS for `JDBCResultSet` was obtained by using CONTRACTOR on the contract-based specification defined in [BBA09] for the `ResultSet` JDBC interface[1]. We validated the contract specification by comparing the preconditions of each method against the conditions that guard exception throwing statements in the reference implementation. Indeed, our analysis concludes that resulting EPA is that of actual protocol LTS for the `JDBCResultSet` reference implementation. Figures 3.1, 3.2, 3.3, 3.4 and 3.5 show the EPAs of the intended protocols of `ListIterator`, `Signature`, `Socket`, `SMTPProcessor` and `JDBCResultSet`[2] respectively.

It is important to note that CONTRACTOR is not specifically designed for constructing EPAs of protocol LTS. When used to produce source code abstractions, CONTRACTOR's input is a not only a class C but also a *requires* clause for each method that is expected to be featured in the EPA. The resulting EPA will be that of a protocol LTS if and only if the requires clause of every method perfectly guards the conditions in which the method is guaranteed not to raise an exception and to terminate. In other words, the requires clauses should capture the failure model described in Section 2.1. Similarly, when used to produce contract-based specification abstractions, the resulting EPA will only be an EPA of the intended protocol LTS described by the contract-based specification if the preconditions for methods adequately capture our failure model.

[1] Available at http://www.cs.cmu.edu/ kbierhof/plural/plural-java-apis.zip

[2] In the EPA of `JDBCResultSet` SIMPLE ACTION SET stands for actions *clearWarnings, getCursorName, getMetaData, findColumn, getFetchDirection, setFetchDirection, setFetchSize, getFetchSize, getType, getConcurrency, rowUpdated, rowInserted, rowDeleted, getStatement, isClosed, getHoldability, getWarnings*. These actions do not make an object to change its state. They were excluded from the figure for readability reasons.

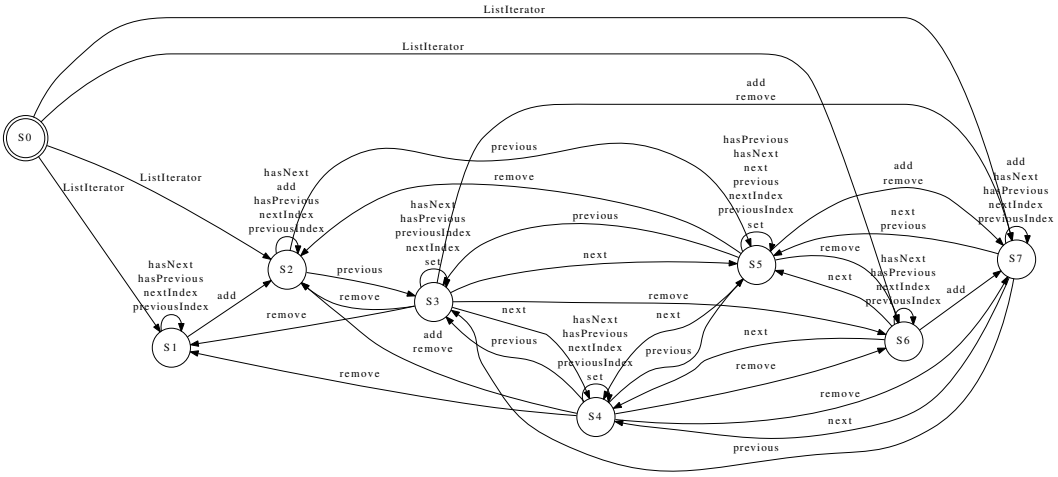


Figure 3.1: EPA of the semantics of `ListIterator` intended protocol

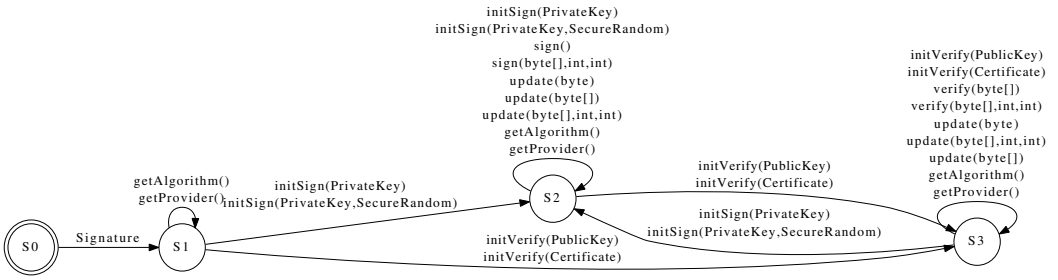


Figure 3.2: EPA of the semantics of `Signature` intended protocol

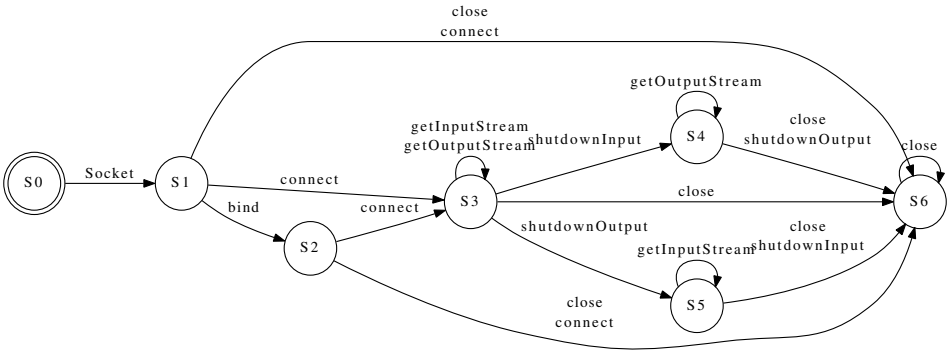


Figure 3.3: EPA of the semantics of `Socket` intended protocol

3.4 Experiment Implementation Details

In this study, for each subject we generated 10000 unit tests by using RANDOOP [PLEB07], an automatic unit test generator for Java classes. It uses a random approach that generates test sequences by randomly choosing method calls. The random sequences generated by Randoop do not necessarily correspond to valid sequences according to the intended protocol of a class. Therefore, we performed simple code instrumentation on subject classes using ASPECTJ in order to guarantee that generated sequences do not fall out of the subjects' call protocol.

For obtaining mutated versions of each subject class we used μ-JAVA [MOK05], a mutation system for Java programs. In order to obtain as many implementations as possible, we let μ-JAVA apply every mutation operator whenever possible. Some of

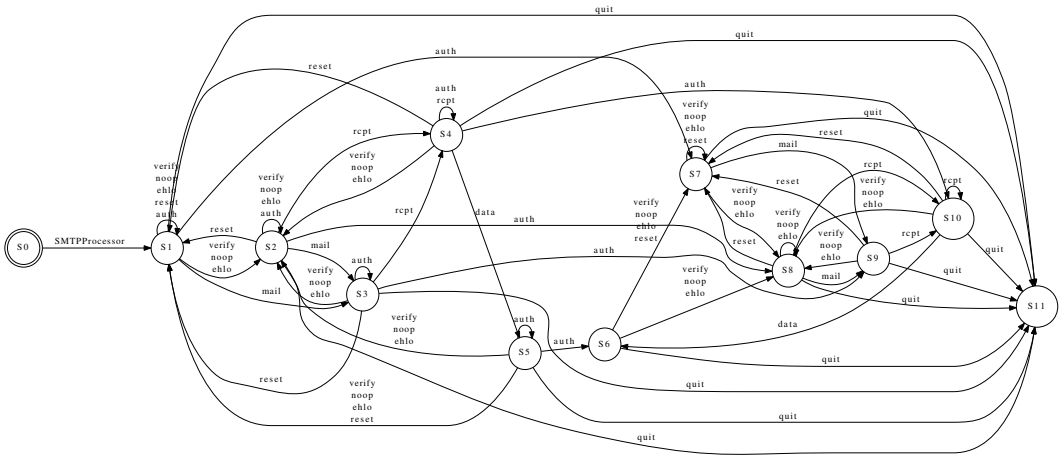


Figure 3.4: EPA of the semantics of `SMTP` intended protocol

these mutants may not be semantically equivalent but there is no evidence that their actual protocol is different to the one of the reference implementation. Although it may not be the case for every non-detected mutant, automatically detecting such equivalences has been proved to be undecidible [OP97]. Due to the large number of mutants generated, a manual inspection for detecting equivalences would have been unfeasible and certainly error-prone. Mutants that were not killed were considered conformant as in [ABLN06].

Our setting is restricted to deterministic failure behaviour of API implementations. This restriction greatly simplifies experimental methodology. To perform a quantitative analysis of the effectiveness of the proposed criteria we must identify which tests kill each mutant (i.e., which sequences of method invocations reveals that an implementation does not behave as expected). If nondeterministic API implementations were included, the notion of test effectiveness (killing a mutant) would require probabilistic treatment since a nondeterministic mutant could behave differently in terms of failing or not to accept a given sequence of calls. Thus, each test would have a probability of killing each mutant. Assuming determinism allows us to perform mutation analysis, which is a common practice approach for empirical assessment of test techniques in software testing research (see for instance [ABLN06, GGZ+13, JH11]). We carefully try to avoid non-deterministic failure behaviour of the subjects through a controlled execution environment. For instance, we run on a single machine the case studies `SMTPProcessor`, `Socket` and `JDBCResultSet` to avoid non-deterministic behaviour resulting from issues in network communication.

We measure statement and branch coverage using Cobertura, a Java tool that calculates the percentage of code accessed by tests. It instruments the class under test and records which lines of code are and are not executed as the test suite runs. It reports on both code and branch coverage.

Table 3.1 summarises relevant information for each subject. Column 2 exhibit lines of code; column 3 the number of generated unit tests; column 4 and 5 show the number of mutants generated and detected respectively; column 6 indicates the number of states of the corresponding EPAs; and columns 7 and 8 show the total and reached (by at least one test suite) number of EPA transtions.

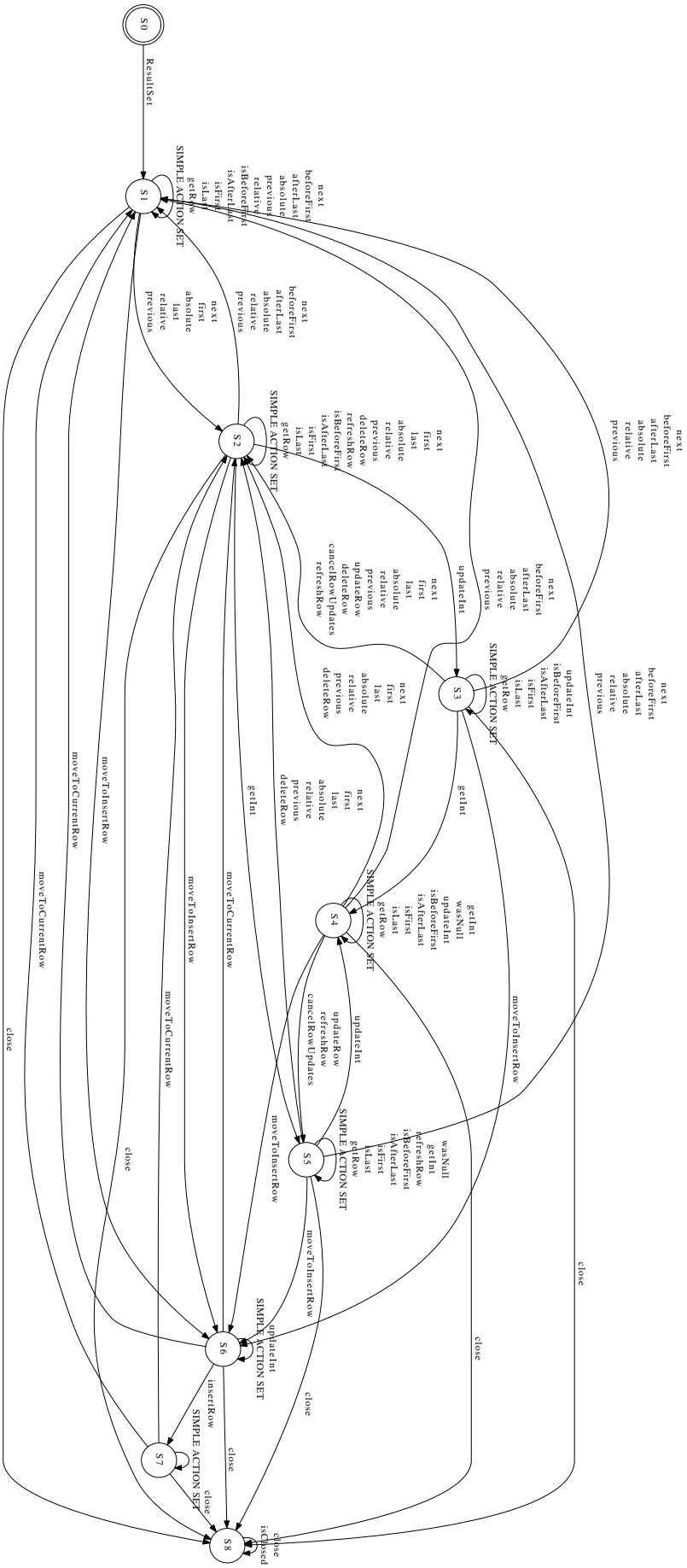


Figure 3.5: EPA of the semantics of `JDBCResultSet` intended protocol

Subject	LOC	Unit Tests	Mutants		EPAs		
			Total	Killed	States	Transitions	
						Total	Covered
`ListIterator`	59	10000	207	145 (70%)	8	68	63 (93%)
`Signature`	121	10000	150	96 (64%)	4	29	27 (93%)
`Socket`	144	10000	98	59 (60%)	7	20	18 (90%)
`SMTPProcessor`	404	10000	570	317 (56%)	12	85	70 (82%)
`JDBCResultSet`	785	10000	404	259 (64%)	9	247	199 (81%)

Table 3.1: Subject classes summary

Part III

Experimental Evaluation

CHAPTER 4

Quantitative Analysis

In this chapter we report on the analysis performed over the measurements obtained in the experiments detailed in Chapter 3 for answering the first research question:

> **RQ.1.** To what extent does the EPA Transition Adequacy Level predict the ability of test suites to detect faults in the context of call protocol conformance testing?

This research question seeks to quantitatively analyse the fault detection ability of EPA transition coverage. This question is addressed in two ways, first looking at correlation between EPA transition coverage and fault detection, and then looking at the impact of EPA transition coverage over fixed-size test suites.

4.1 The Criterion as Fault Detection Predictor

> **RQ.1.1.** Does the EPA Transition Adequacy Level have a high correlation with fault detection in the context of protocol conformance testing?

In order to determine to what extent the EPA Transition Adequacy Level of definition 6 predicts the ability of test suites to detect faults that manifests themselves as protocol conformance failures we use Spearman's rank correlation coefficient ρ. The coefficient measures how well the relationship between two variables can be described by a monotonic function. The coefficient lies in the interval [-1,1]. Values near 1 and -1 signal a strong dependence between both variables (direct and inverse, respectively), while values around 0 indicate no dependence between them. We choose the Spearman's coefficient because unlike others commonly used, such as Pearson product-moment coefficient, it does not make any assumption about the distribution of data. This is important as we were unable to establish that data fitted known distributions using goodness of fit tests such as Kolmogorov-Smirnov.

We compute the coefficient not only to correlate EPA Transition Adequacy Level against detected faults, but also for correlating statement and branch coverage

against detected faults. The coefficients provide us a measure of statistical dependence between the degree of coverage of each criterion and the number of detected faults by a test suite. Table 4.1 shows the ρ values obtained for each criterion.

Subject	Stmt Coverage	Brch Coverage	Tx Coverage
`ListIterator`	0.48	0.61	**0.84**
`Signature`	0.65	0.71	**0.73**
`Socket`	0.81	**0.86**	0.72
`SMTPProcessor`	0.66	**0.78**	0.69
`JDBCResultSet`	0.89	**0.92**	0.68

Table 4.1: Correlation between coverage and fault detection. Bold indicates the highest value for a row.

Transitions coverage has high ($\rho > 0.7$) or very close to high correlation for all cases. On the other hand, white-box criteria correlation values are more disperse indicating only moderate correlation in some subjects. More precisely,for `ListIterator` and `Signature` the highest correlation is given by EPA coverage; while for `Socket`, `SmtpProcessor` and `JDBCResultSet` by branch coverage. It is worth noting that, whereas for all subjects the correlation of EPA coverage are around 0.7 and their values are steady, for both statement and branch coverage there are subjects for which the correlation is moderate and their values appear to be more unstable. Besides, while in order to use the code coverage criteria source code is needed, our criterion only requires a specification from which the EPA could be obtained.

Two case studies illustrate the best and worst cases for transition coverage and structural coverage criteria. On the one hand, comparatively worse performance of transition criterion on `JDBCResultSet` can be explained by the unbalanced API implementation of the subject: There is one method out of forty-two that collects the 35% of all mutations of the class. For killing these mutants a test should execute that method (and consequently its α-abstracted execution traverses a few sepecific transitions of the 247 of the EPA). In other words, for killing a large number of the mutants traversing a lot of transitions do not help; in these cases a specific transition shoud be exercised. On the other hand, for `ListIterator` transition coverage has a high correlation compared to that of statement and branch coverage. This can be explained by the fact that the subject presents a simple code structure (no loops, few branches) within methods that can be easily covered by test suites. However, the structure of the intended protocol LTS, and hence its EPA, is quite rich. Achieving coverage of the EPA requires executing more complex sequences of method calls that explore interesting states of the iterator (such as getting to the end of the list). High code coverage does not guarantee reaching such states.

As can be seen, black-box EPA transition coverage has high correlation with bug detection. In some subjects transition coverage seems to be a better proxy of excercised concrete behavior than white-box criteria. Since EPA transtion coverage privileges diversity of transtions regardless whether or not actions are covered, correlation with bug finding is reduced when a particular action is error prone.

4.2 Hypothesis Tests for Fixed-Size Test Suites

RQ.1.2. Given a fixed test suite size, do test suites with higher EPA Transition Adequacy Level perform better in terms of fault detection than those with lower EPA Transition Adequacy Level?

The impact of size (as the number of calls to the software under test) on the effectiveness of test suites has been addressed by several works [NA09, AGWX08]. Although it is known that not only the size determines the quality of a test suite, in general it is expected that the likelihood of revealing faults increases with the number of invocations featured by a test suite [NA09].

The tests that achieve higher EPA coverage are generally also longer. Thus, in order to refute that EPA coverage correlates with fault detection only to required test suite size, we analyse results on a "by size" basis: given a size we aim to study if the test suites with higher EPA Transition Adequacy Level are likely to detect more faults than those with lower adequacy when fixing the size of test suites.

Samples for each size are not large enough for obtaining statistically significant results. Therefore, we divide them into bins grouping those of similar size in the same bin. For each subject, we ensure that the difference of size between test suites of the same bin do not exceed 10% of the difference of size between the largest and the smallest overall test suites.

We want to show that given a bin, picking a test suite from the best test suites is more likely to give a test suite with higher fault detection ability than picking it from the rest of the test suites of that bin.

We define the set of test suites in a bin with higher adequacy as those that achieve at least 80% of the coverage that is achieved by the test suite with highest coverage of that bin. This is because the degree of coverage varies from bin to bin due to the change in test suite sizes. In this way we can obtain the "best" test suites for a particular bin.

As explained in Section 4.1, detected fault data does not necessarily fit standard distributions. Therefore, as before, we choose a nonparametric test for our analysis. In order to compare higher coverage test suites of a bin against the rest of that bin, we use the Mann-Whitney U test, a non-parametric statistical hypothesis test for assessing whether the probability of an observation from one population exceeding an observation from a second population is not equal to 0.5. This hypothesis test assumes that all the observations from both groups are independent of each other, which is true because the test suites that do fit our criterion and those that do not form disjoint sets. It also requires the responses are ordinal or continuous measurements, which is also true because the variables considered here are EPA Transition Adequacy Level and detected faults. Under the null hypothesis the probability of a random observation from one population P_1 exceeding a random observation from the second population P_2 equals the probability of an observation from P_2 exceeding an observation from P_1. Under the alternative hypothesis the probability is not equal to 0.5. That is, values from one population tend to exceed those of the other. We reject the null hypothesis when the ρ-value resulting from the hypothesis test is less than 0.05. In our case, rejecting the null hypothesis means that tests with high transition coverage are better in terms of fault detection than those of low coverage, and that that is not just due to chance.

To also assess the magnitude of the improvement we use the Vargha and Delaney's A_{12} effect size (ES). This non-parametric measure has recently been advo-

cated in [AB11] for randomized algorithms. In our case, in a given bin A_{12} estimates the probability that choosing a tests suite of high transition coverage detects more mutants than one chosen randomly from the population of low coverage. We also report the confidence interval (CI) for the effect size stated at the 95% confidence level.

Table 4.2 shows the test results for all subjects. Each row of the table corresponds to one bin. The second column specifies the test suite size interval of each bin. The third indicates the minimum and maximum number of EPA transitions covered by them, and the fourth shows the threshold number of transitions that a test suite must cover to be considered adequate (i.e. coverage of at least 80% of the best coverage in the bin). The fifth and sixth columns show the number tests suites that are adequate (i.e. highest coverage) and not adequate. The seventh column exhibits the ρ-value of the Mann-Whitney test, the eighth the A_{12} effect size and the ninth its confidence interval.

Subject	EPA Coverage					Fault Detection		
	Size	Cov Tx	Thr	#TS$_{\geqslant \text{Thr}}$	#TS$_{<\text{Thr}}$	p	ES	CI
ListIterator	[50,99]	16 - 37	30	90	222	~ 0	0.73	[0.67,0.79]
	[100,149]	22 - 44	36	68	219	~ 0	0.81	[0.75,0.86]
	[150,199]	24 - 48	39	66	215	~ 0	0.72	[0.66,0.79]
	[200,249]	29 - 48	39	163	135	~ 0	0.72	[0.66,0.77]
	[250,299]	32 - 50	40	189	113	~ 0	0.72	[0.66,0.78]
	[300,349]	34 - 51	41	203	89	~ 0	0.68	[0.62,0.74]
	[350,399]	33 - 51	41	197	44	~ 0	0.75	[0.67,0.83]
	[400,449]	35 - 52	42	161	26	$0.08^{\dagger}$	0.61	[0.51,0.72]
	[450,499]	36 - 53	43	145	28	~ 0	0.73	[0.63,0.82]
	[500,549]	36 - 52	42	113	14	$0.08^{\dagger}$	0.63	[0.49,0.77]
Signature	[50,99]	9 - 22	18	98	189	~ 0	0.80	[0.71,0.90]
	[100,149]	11 - 25	20	114	110	~ 0	0.75	[0.64,0.86]
	[150,199]	12 - 26	21	110	130	~ 0	0.77	[0.66,0.87]
	[200,249]	14 - 26	21	165	79	~ 0	0.77	[0.66,0.88]
	[250,299]	14 - 27	22	181	63	~ 0	0.83	[0.71,0.95]
	[300,349]	14 - 27	22	201	43	~ 0	0.85	[0.73,0.97]
	[350,399]	16 - 27	22	232	83	~ 0	0.75	[0.64,0.86]
	[400,449]	18 - 27	22	205	35	~ 0	0.90	[0.80,1.00]
	[450,499]	17 - 27	22	220	31	~ 0	0.82	[0.70,0.94]
	[500,549]	17 - 27	22	197	14	0.01	0.86	[0.79,0.93]
Socket	[0,12]	2 - 9	8	29	141	~ 0	0.77	[0.70,0.84]
	[13,25]	5 - 13	11	36	261	~ 0	0.72	[0.64,0.79]
	[26,38]	5 - 14	12	64	227	~ 0	0.69	[0.63,0.76]
	[39,51]	7 - 16	13	73	227	0.01	0.60	[0.55,0.66]
	[52,64]	8 - 16	13	151	167	~ 0	0.59	[0.56,0.63]
	[65,77]	9 - 16	13	206	91	0.03	0.58	[0.54,0.62]
	[78,90]	10 - 17	14	170	143	$1.00^{\dagger}$	0.50	[0.48,0.52]
	[91,103]	10 - 17	14	178	106	$0.37^{\dagger}$	0.53	[0.51,0.55]
	[104,116]	10 - 16	13	159	18	0.03	0.63	[0.54,0.72]
	[117,129]	11 - 16	13	45	8	$1.00^{\dagger}$	0.40	[0.40,0.40]
SMTPProcessor	[0,149]	4 - 28	23	12	184	~ 0	0.80	[0.68,0.93]
	[150,299]	11 - 37	30	21	242	0.03	0.65	[0.56,0.75]
	[300,449]	20 - 38	31	86	161	~ 0	0.63	[0.57,0.70]
	[450,599]	21 - 40	32	122	137	~ 0	0.61	[0.55,0.67]
	[600,749]	24 - 45	36	104	150	~ 0	0.62	[0.56,0.68]
	[750,899]	26 - 46	37	114	136	0.01	0.61	[0.55,0.67]
	[900,1049]	28 - 50	40	96	186	~ 0	0.64	[0.58,0.70]
	[1050,1199]	31 - 48	39	171	97	0.02	0.59	[0.53,0.65]
	[1200,1349]	31 - 49	40	153	105	$0.13^{\dagger}$	0.55	[0.49,0.62]
	[1350,1499]	30 - 50	40	168	55	$0.45^{\dagger}$	0.54	[0.46,0.61]
JDBCResultSet	[200,399]	69 - 107	86	75	81	~ 0	0.70	[0.62,0.79]
	[400,599]	80 - 114	92	213	56	~ 0	0.74	[0.66,0.81]
	[600,799]	90 - 123	99	258	25	~ 0	0.78	[0.67,0.88]
	[800,999]	94 - 134	108	231	32	$0.10^{\dagger}$	0.59	[0.48,0.70]
	[1000,1199]	102 - 143	115	172	63	0.02	0.62	[0.53,0.70]
	[1200,1399]	108 - 140	112	246	8	$0.73^{\dagger}$	0.41	[0.21,0.61]
	[1400,1599]	106 - 140	112	246	11	$0.09^{\dagger}$	0.73	[0.69,0.76]
	[1600,1799]	107 - 151	121	252	19	0.01	0.69	[0.58,0.80]
	[1800,1999]	119 - 153	123	230	17	$0.86^{\dagger}$	0.47	[0.21,0.74]
	[2000,2199]	120 - 153	123	248	17	$0.06^{\dagger}$	0.70	[0.57,0.82]

Table 4.2: Mann-Whitney test results for fault detection. Values not statistically significant are marked with †.

Results indicate that EPA coverage makes a difference in terms of fault detection for tests suites of the same size. Considering all the five case studies, in 39 out of 50 bins (78% of the bins) there is statistical evidence that test suite with higher coverage of EPA transitions perform better than those of lower coverage (column 7). Moreover, except for two bins of `JDBCResultSet`, those on which there is no

statistical evidence correspond to bins containing very large test suites. That makes sense, since the impact of coverage tends to decrease as the size of test suites increases. For many of these bins the sample size of test suites that do not reach the coverage threashold is quite small (ranging between 8 and 28 individuals in five of these bins), difficulting the possibilty of obtaining statistical conclusions. The case of the bin corresponding to largest test suites of `JDBCResultSet` illustrates this situation: although the effect size is 0.70, the number of test suites that do and do not reach the coverage threashold are 248 and 17 respectively. In this case the ρ-value is 0.06. Consequently, according to the significance level chosen in our experiments we cannot reject the null hypothesis in this case.

4.3 Summary of Results

Results suggest that the EPA transition criterion is a good predictor of mutant detection (RQ.1.1) with correlations between 0.68 and 0.84 depending on the case study. Results also show that EPA transition correlations are comparable or better than those of structural (rather than behavioural) white box criteria. To avoid benefitting from bias in the selection of the specification language and model to be structurally covered, we select the (unmutated) code itself as the specification. We believe that this choice does not favour (on the contrary) the hypotheses we propose in this thesis as the code can be considered as the most detailed specification and most likely constitutes an upper bound on what structural criteria on specifications can achieve as test suite quality predictors. Results suggest that the EPA transition coverage criterion, which is behavioral and hence independent of specification language bias, performs comparably in terms of predictability of test suite fault detection, resulting in higher values for all case studies against statement coverage and is better than branch coverage in two out of five case studies.

Results also suggest that for fixed-size, test suites with the highest behavioral adequacy are statistically better (RQ.1.2). In all the five case studies we divide the test suites in ten different bins grouping those of similar size. We perform the Mann-Whitney U hypothesis test and observe that in 39 out of 50 bins there is statistical evidence that those of higher adequacy perform better. Those for which no such evidence exists correspond to bins of the largest test suites. As expected, the impact of EPA transition coverage diminishes as size increases. Additionally, to get a quantitative measure of this fact we calculate the Vargha- Delaney A_{12} effect size, which measures the difference between two populations in terms of the probability that a score sampled at random from the first population will be greater than a score sampled at random from the second.

CHAPTER 5

Qualitative Analysis

In this chapter we address the second research question. It aims at providing more qualitative results that may provide insight on why positive results for RQ.1 are obtained.

> **RQ.2.** Why does EPA Transition Adequacy Level predict the ability of test suites to detect faults that manifest themselves as call protocol conformance failures?

We analize this research question from three different perspectives:

1. correlation between EPA transition coverage and code coverage (*RQ.2.1*);
2. the characteristics of category partition criterion implicitly defined by EPA transition coverage (an input i belongs to the subdomain defined by a transition t if and only if i exercies t) (*RQ.2.2*); and
3. how faults are detected by the traversal of EPA transitions (*RQ.2.3*).

First, we show that covering EPA transitions correlates with structural coverage. Again, we also perform the analysis in a "by size" basis and see that those test suites with higher adequacy according to the EPA coverage criterion perform better than the rest in terms of structural coverage. Then we study the properties of the subdomains implicitly derived from the criterion from a category partition perspective and see that resulting partition tends to generate at least one subdomain dense in faults. Finally, we study the relation between covering transitions and fault detection, and observe that for each fault there is almost always one transition highly effective in detecting it, while nearly all the rest have poor effectiveness.

5.1 The Criterion as Structural Coverage Predictor

> **RQ.2.1.** Does EPA Transition Adequacy Level predict the statement and branch coverage?

To measure correlation between EPA transition coverage and code coverage, we first investigate if the code coverage achieved by test suites fits a standard distribution. As with the relation between EPA coverage and fault detection, this is not the case. Consequently, to find out how well the coverage of EPA predicts code coverage we again calculate Spearman's rank correlation coefficient ρ. The results are shown in table 5.1.

Class	Tx/Stmt Corr	Tx/Brch Corr
`ListIterator`	0.66	**0.70**
`Signature`	0.63	**0.70**
`Socket`	0.77	**0.77**
`SMTPProcessor`	0.70	**0.73**
`JDBCResultSet`	**0.67**	0.66

Table 5.1: Correlation between EPA transitions and structural coverage. Bold indicates the highest value for a row.

Results lead to believe that the EPA coverage criterion is a reasonably good predictor of statement and branch coverage adequacy. Correlation against branch coverage is moderate to high depending on the case and values are consistently close to 0.7. Interestingly, except for transition coverage in `JDBCResultSet`, in all cases the correlation with branch coverage is greater or equal than that of statement coverage. In the type of software under analysis the selection of branches of conditional expressions is often based on the internal state of the object. Thus, the high correlation values may be an indication that indeed EPA states capture important properties of the object states, and therefore may be useful to use them for abstracting concrete object states. It is worth noting that in almost all cases the EPA transition coverage criterion works better as predictor of mutant detection than as code coverage predictor -which seems consistent to the rationale underlying criteria.

Similarly to RQ.1.2, tests that achieve higher EPA coverage are generally also larger, and also larger tests tend to cover larger a portion of the code. Thus, we analize the relation between EPA and code coverage in a "by size" basis using the same set of bins as in RQ.1.2. Again, we do not assume code coverage achieved by test suites fits a common distributions. Thus, in order to determine if adequate tests achieve higher code coverage than non-adequate ones we perform Mann-Whitney U tests. As for fault detection, the assumptions made by the test are also met in this case. Again, we reject the null hypothesis when the ρ-value resulting from the hypothesis test is less than 0.05. In this case, rejecting the null hypothesis means that tests with high transition coverage are better in terms of code coverage than those of low coverage, and that it is not just by cause of chance. We also calculate the A_{12} effect size and its confidence interval. Given a bin, A_{12} estimates the probability that choosing a test suite of high transition coverage has higher statement/branch coverage than a test suite chosen randomly from the population of low transition coverage. We also report the confidence interval (CI) for the effect size stated at the 95% confidence level.

Table 5.2 shows the results for all subjects. Each row of the table corresponds to one bin. The second column specifies the test suite size interval of each bin. The third indicates the minimum and maximum number of EPA transitions covered by them, and the fourth shows the threshold number of transitions that a test suite must cover to be considered adequate (those that cover at least 80% of the best

coverage in the bin). The fifth and sixth columns show the number tests suites that are adequate (i.e. highest coverage) and not adequate. Columns seven to nine exhibit the ρ-value of the Mann-Whitney test, the A_{12} effect size and its confidence interval respectively for statement coverage. And columns ten to twelve show these values for branch coverage.

Subject	EPA Coverage					Statement Coverage			Branch Coverage		
	Size	Cov Tx	Thr	$\#TS_{\geqslant Thr}$	$\#TS_{<Thr}$	p	ES	CI	p	ES	CI
ListIterator	[50,99]	16 - 37	30	90	222	$0.23^{\dagger}$	0.54	[0.51,0.58]	~ 0	0.74	[0.69,0.79]
	[100,149]	22 - 44	36	68	219	$0.66^{\dagger}$	0.52	[0.50,0.53]	~ 0	0.79	[0.75,0.82]
	[150,199]	24 - 48	39	66	215	$0.90^{\dagger}$	0.50	[0.50,0.51]	~ 0	0.70	[0.67,0.74]
	[200,249]	29 - 48	39	163	135	$1.00^{\dagger}$	0.50	[0.50,0.50]	~ 0	0.62	[0.58,0.66]
	[250,299]	32 - 50	40	189	113	$0.93^{\dagger}$	0.49	[0.49,0.50]	~ 0	0.66	[0.61,0.70]
	[300,349]	34 - 51	41	203	89	$1.00^{\dagger}$	0.50	[0.50,0.50]	0.02	0.58	[0.54,0.63]
	[350,399]	33 - 51	41	197	44	$1.00^{\dagger}$	0.50	[0.50,0.50]	$0.13^{\dagger}$	0.58	[0.52,0.63]
	[400,449]	35 - 52	42	161	26	$1.00^{\dagger}$	0.48	[0.48,0.48]	$0.08^{\dagger}$	0.62	[0.54,0.70]
	[450,499]	36 - 53	43	145	28	$1.00^{\dagger}$	0.48	[0.48,0.48]	0.03	0.62	[0.54,0.70]
	[500,549]	36 - 52	42	113	14	$1.00^{\dagger}$	0.50	[0.50,0.50]	0.02	0.72	[0.60,0.85]
Signature	[50,99]	9 - 22	18	98	189	~ 0	0.76	[0.66,0.85]	~ 0	0.78	[0.68,0.88]
	[100,149]	11 - 25	20	114	110	~ 0	0.76	[0.65,0.87]	~ 0	0.74	[0.63,0.85]
	[150,199]	12 - 26	21	110	130	0.01	0.70	[0.60,0.80]	~ 0	0.80	[0.70,0.89]
	[200,249]	14 - 26	21	165	79	~ 0	0.73	[0.61,0.85]	~ 0	0.80	[0.70,0.90]
	[250,299]	14 - 27	22	181	63	~ 0	0.73	[0.62,0.84]	~ 0	0.81	[0.70,0.92]
	[300,349]	14 - 27	22	201	43	~ 0	0.80	[0.67,0.92]	~ 0	0.83	[0.72,0.95]
	[350,399]	16 - 27	22	232	83	0.02	0.67	[0.56,0.77]	~ 0	0.75	[0.65,0.85]
	[400,449]	18 - 27	22	205	35	0.01	0.73	[0.61,0.85]	0.01	0.77	[0.63,0.90]
	[450,499]	17 - 27	22	220	31	$0.16^{\dagger}$	0.65	[0.50,0.80]	~ 0	0.84	[0.69,0.98]
	[500,549]	17 - 27	22	197	14	0.01	0.83	[0.76,0.90]	0.01	0.77	[0.67,0.86]
Socket	[0,12]	2 - 9	8	29	141	~ 0	0.86	[0.77,0.94]	~ 0	0.82	[0.74,0.91]
	[13,25]	5 - 13	11	36	261	~ 0	0.89	[0.84,0.94]	~ 0	0.83	[0.77,0.89]
	[26,38]	5 - 14	12	64	227	~ 0	0.78	[0.72,0.83]	~ 0	0.74	[0.68,0.80]
	[39,51]	7 - 16	13	73	227	~ 0	0.71	[0.66,0.76]	~ 0	0.70	[0.64,0.76]
	[52,64]	8 - 16	13	151	167	~ 0	0.67	[0.62,0.71]	~ 0	0.66	[0.61,0.70]
	[65,77]	9 - 16	13	206	91	~ 0	0.67	[0.62,0.72]	~ 0	0.68	[0.63,0.73]
	[78,90]	10 - 17	14	170	143	0.01	0.59	[0.55,0.62]	~ 0	0.59	[0.55,0.63]
	[91,103]	10 - 17	14	178	106	0.01	0.59	[0.55,0.63]	0.01	0.59	[0.55,0.63]
	[104,116]	10 - 16	13	159	18	~ 0	0.72	[0.61,0.82]	~ 0	0.71	[0.61,0.81]
	[117,129]	11 - 16	13	45	8	$0.63^{\dagger}$	0.33	[0.29,0.38]	$0.30^{\dagger}$	0.26	[0.12,0.39]
SMTPProcessor	[0,149]	4 - 28	23	12	184	~ 0	0.86	[0.78,0.94]	~ 0	0.88	[0.80,0.96]
	[150,299]	11 - 37	30	21	242	0.02	0.66	[0.57,0.75]	0.01	0.68	[0.59,0.77]
	[300,449]	20 - 38	31	86	161	~ 0	0.65	[0.59,0.72]	~ 0	0.65	[0.58,0.72]
	[450,599]	21 - 40	32	122	137	0.03	0.59	[0.52,0.65]	0.04	0.57	[0.51,0.64]
	[600,749]	24 - 45	36	104	150	~ 0	0.61	[0.55,0.68]	~ 0	0.62	[0.55,0.68]
	[750,899]	26 - 46	37	114	136	0.01	0.60	[0.54,0.66]	~ 0	0.62	[0.56,0.69]
	[900,1049]	28 - 50	40	96	186	0.02	0.59	[0.53,0.65]	~ 0	0.62	[0.56,0.68]
	[1050,1199]	31 - 48	39	171	97	$0.20^{\dagger}$	0.55	[0.49,0.61]	0.03	0.58	[0.52,0.65]
	[1200,1349]	31 - 49	40	153	105	0.01	0.61	[0.54,0.67]	~ 0	0.62	[0.56,0.68]
	[1350,1499]	30 - 50	40	168	55	$0.50^{\dagger}$	0.53	[0.45,0.61]	$0.07^{\dagger}$	0.58	[0.50,0.66]
JDBCResultSet	[200,399]	69 - 107	86	75	81	~ 0	0.66	[0.57,0.74]	~ 0	0.66	[0.57,0.75]
	[400,599]	80 - 114	92	213	56	~ 0	0.72	[0.64,0.80]	~ 0	0.68	[0.60,0.76]
	[600,799]	90 - 123	99	258	25	0.01	0.68	[0.58,0.79]	~ 0	0.72	[0.60,0.83]
	[800,999]	94 - 134	108	231	32	0.04	0.55	[0.44,0.66]	0.04	0.53	[0.43,0.64]
	[1000,1199]	102 - 143	115	172	63	0.02	0.61	[0.52,0.70]	0.04	0.57	[0.48,0.65]
	[1200,1399]	108 - 140	112	246	8	0.03	0.33	[0.19,0.46]	0.04	0.39	[0.17,0.61]
	[1400,1599]	106 - 140	112	246	11	$0.09^{\dagger}$	0.76	[0.72,0.79]	$0.10^{\dagger}$	0.68	[0.65,0.72]
	[1600,1799]	107 - 151	121	252	19	~ 0	0.76	[0.67,0.85]	~ 0	0.72	[0.64,0.81]
	[1800,1999]	119 - 153	123	230	17	$0.49^{\dagger}$	0.53	[0.40,0.67]	$0.94^{\dagger}$	0.45	[0.27,0.62]
	[2000,2199]	120 - 153	123	248	17	0.01	0.79	[0.68,0.89]	$0.08^{\dagger}$	0.65	[0.57,0.73]

Table 5.2: Mann-Whitney test results for code coverage. Values not statistically significant are marked with †.

Again, results show that in general there is statistical significant evidence for saying that test suites achieving higher EPA coverage adequacy are more likely to achieve higher code coverage than the general population of a given size. Considering the five case studies, there is such evidence in 34 out of 50 bins (68% of the bins) in the case of statement coverage. Interestingly, 10 of the bins for which there is no evidence are the bins of `ListIterator`. As explained before, the simplicity of its code makes it easy to achieve high code coverage, and therefore covering many transitions does not make a significant difference (but it does affect positively fault detection, as seen). For `Socket` and `Signature` the evidence exists in 8 out of 10 bins, and for `JDBCResultSet` and `SMTPProcessor` in 9 out of 10. As regards branch coverage, in 43 out of 50 bins (86% of the bins) there is statistical evidence of the convenience of tests with high EPA coverage.

As with fault detection, those bins for which there is no statistical evidence correspond to very large test suites (with the exception of `ListIterator`) which is

consistent with the fact that the impact of high coverage tends to decrease as the size of the test suite increases.

Summarizing, results indicate that EPA transition coverage has high correlation with branch coverage (except for the case of `JDBCResultSet`, which is quite close to 0.7 anyway), and a little lower correlation with statement coverage (but still with values close to 0.7 in all cases). In addition, for fixed-size test suites there is statistical evidence that in most cases those tests with higher transition coverage perform better in terms of structural coverage.

5.2 A Category Partition Perspective

RQ.2.2. Does the category partition implicitly defined by EPA transition coverage result in subdomains with high failure rates?

Category partition testing refers to a general family of testing strategies [JW89]. They consist in dividing the program's input domain into subdomains in such a way that they span the domain's input space. Then the program is tested selecting some inputs from each subdomain. Ideally, each subdomain should be such that the program under test behaves as expected for every element or fails for every element. When all the inputs of a subdomain cause the program to fail the subdomain is called *100% revealing* or simply *revealing* [WO80]. The purpose is to make the partition in a way that the resulting test set results in a good representation of the whole program's domain.

As an example, given a program P with domain $\mathbb{Z}$, we may consider the subdomains $\mathbb{Z}^-$, $\{0\}$ and $\mathbb{Z}^+$ (negative integers, zero and positive integers). In this case we are dividing the input domain into *disjoint* subdomains. However, the partition may also result in *overlapping* subdomains. Let us illustrate this with the subdomains derived from the statements of a program. They divide the input domain into non-disjoint subdomains where each subdomain consists of all test cases which cause a particular statement to be executed. A test case causes many statements to be executed and consequently is a member several subdomains.

An important *analytical* study on partition testing is presented in [JW89]. The paper compares partition testing against random selection of inputs and shows the conditions under which one of them outperforms the other. It concludes that a partition testing strategy is neither good nor bad per se. Its effectiveness lies in how the domain is divided into subdomains. More precisely, it depends on how the inputs which cause a program to fail are concentrated within the subdomains defined by the partition. As above mentioned, an ideal scenario contains at least one revealing subdomain. No matter which input is selected from it, the program will fail and be revealed as incorrect. Of course that such situation is not realistic: when testing a program the location of faults is unknown and there is no guarantee that one subdomain contains only error-revealing inputs. On the other hand, when only one or a few elements of a subdomain make a program fail then the density of bugs in the relevant subdomains will be relatively low, and therefore the strategy of picking from all subdomains will be of low effectiveness. What is more, under the latter circumstances it may be the case that the probability of detecting a fault when using category partition is lower than the one resulting of a random selection of inputs. Still, if there is at least one subdomain that is dense in failures -i.e., that most of its inputs are error-revealing-, then the approach is effective [JW89].

The proposed coverage criterion implicitly divides the input's domain of an intended protocol LTS according to the transitions of its associated EPA. In that case, a unit test u belongs to the category defined by a transition t if and only if the α-abstracted execution of u exercises t. As with statements, the subdomains defined by transitions of an EPA may overlap. For instance, the test of listing 2.2 belongs to the subdomains derived from transitions (S_0, push, S_1) and (S_1, pop, S_0), as shown in figure 2.4.

In order to determine if there is a subdomain dense for a faulty implementation I we must know the failure rate (FR) of each subdomain. That is, the proportion of inputs that are error-revealing for I. Or what is the same, the probability p of a randomly-chosen input of each subdomain to be error-revealing. In the case of finite domains the calculation is quite simple, since it corresponds to the ratio between error-revealing inputs and the domain size. In general -and in particular in the selected subjects- the input domain space for protocols is infinite. Therfore, for each case study we estimate p by $\hat{p}$ based on experimental observations as follows:

1. for each EPA transition t we calculate the sample size of the subdomain d_t derived from it (i.e., the number of unit tests whose α-abstracted execution traverses t);
2. for each faulty implementation I and subdomain d_t we count the number of error-revealing inputs of the sample $er_{d_t,I}$;
3. for each faulty implementation I and subdomain d_t we calculate $\hat{p}_{d_t,I} = \frac{er_{d_t,I}}{|d_t|}$

Since we are interested in determining if there is at least one subdomain dense in failures, for each faulty implementation I we select $\hat{p}_{d_j,I}$, where d_j is the densest subdomain for I. As a reference, we also provide the density of the entire domain.

It is worth noting that not all subdomain sample sizes are large enough in order to obtain statistically significant results. For instance, in some cases there are EPA transitions reached by only one unit test α-abstracted execution. That is, there are transitions t for which we generated only one input such that its $\alpha-$abstracted execution traverses t. In this case the estimator $\hat{p}_{d_t,I}$ would be equal to 1 or would be equal to 0 depending on whether the unit test kills the mutant I or not (and in the first case, we would be considering that there is a *revealing subdomain* for I). In any case, a sample of size 1 is too small to consider $\hat{p}_{d_t,I}$ as a good estimator of $p_{d_j,I}$. It would be misleading to consider such input as representative of the entire subdomain d_j.

Because the observations are independent, the estimator $\hat{p}$ has a binomial distribution. The maximum variance of this distribution is $0.25 * |d_t|$, which occurs when the real parameter is $p = 0.5$. Since p is unknown, the maximum variance is used for sample size assessments. For sufficiently large $|d_t|$, the distribution of $\hat{p}$ will be closely approximated by a normal distribution. Using this approximation, around 95% of this distribution's probability lies within 2 standard deviations of the mean. Using the Wald method for the binomial distribution, an interval of the form $(\hat{p} - 2\sqrt{0.25/|d_t|}, \hat{p} + 2\sqrt{0.25/|d_t|})$ will form a 95% confidence interval for the true proportion. If this interval needs to be no more than W units wide, the equation $4\sqrt{0.25/|d_t|} = W$ can be solved for $|d_t|$, yielding $|d_t| = 4/W^2 = 1/B^2$ where B is the error bound on the estimate. We fix $B = 5\%$, and therefore we require samples sizes greater or equal to 400 to consider them representative. Table 5.3 shows for each subject the total number of subdomains (or transitions) and how many we filter. Note that filtering subdomains does not favour us. On the contrary, it could be the case that there is a subdomain for which there are not enough inputs whose real failure rate is greater than that of those of the subdomains we are not filtering.

Subject	Total	Filtered
`ListIterator`	68	14 (20.59%)
`Signature`	29	2 (6.90%)
`Socket`	20	4 (20.00%)
`SMTPProcessor`	85	35 (41.18%)
`JDBCResultSet`	247	87 (35.22)

Table 5.3: Total and filtered subdomains

Figure 5.1 shows boxplots of the failure rates of both, the densest subdomains with statistcally significant rates -i.e., for which there are at least 400 inputs- and the entire domain. Numerical values of percentils, means and means excluding outliers are shown in table 5.4.

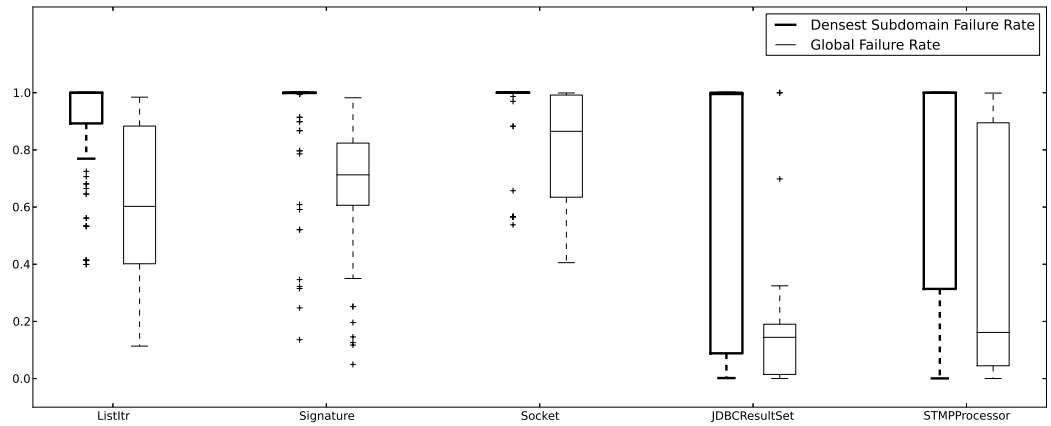


Figure 5.1: Failure rates of the densest subdomain and the entire domain

Subject	By Subdomains					Entire Domain				
	Q_1	Med	Q_3	Mean	Mean EO	Q_1	Med	Q_3	Mean	Mean EO
`ListIterator`	0.89	1.0	1.0	0.91	0.95	0.4	0.6	0.88	0.61	0.61
`Signature`	1.0	1.0	1.0	0.93	0.98	0.61	0.72	0.83	0.67	0.7
`Socket`	1.0	1.0	1.0	0.94	0.99	0.56	0.87	0.99	0.8	0.8
`JDBCResultSet`	0.09	1.0	1.0	0.67	0.67	0.01	0.14	0.19	0.14	0.13
`SMTPProcessor`	0.31	1.0	1.0	0.71	0.71	0.05	0.16	0.86	0.4	0.4

Table 5.4: Statistical info of failure rates considering subdomains and the entire domain

It is noteworthy that in the five case studies, for at least 50% of the non-conformant implementations the division into subdomains according to EPA transitions produces a revealing subdomain. What is more, in the cases of `Signature` and `Socket` the same observation is valid for at least 75% of the faulty implementations; and in the case `ListIterator` for 75% of versions that are not conformant there is a subdomain with density 0.89 or greater. Although it does not happen the same in the cases of `JDBCResultSet` and `SMTPProcessor`, these have in common a significant increase of density with respect to the density of the entire domain. As above mentioned half of the faulty implementations have a revealing subdomain for `JDBCResultSet` and `SMTPProcessor`, but when considering the entire domain the numbers dramatically decrease: their medians are 0.14 and 0.16 respectively (i.e., in both cases there is a difference around 85%).

It can also be observed the difference between subdomains failure-rate mean

values and entire domain mean values: they range between 0.14 (for `Socket`) and 0.53 (for `JDBCResultSet`). In the case of `Socket` the difference is small. This is explained by the (non-)difficulty of the mutants generated by μ-JAVA for this class: all non-conformant implementations are discoverd by at least 40% of the unit tests and 65% of them are killed by 85% of the unit tests or more. That is, all of them are easy to discover, and therefore the failure rates considering the entire domain are also high (mean values removing outliers present only little differences).

The phenomenon described above does not occur in the other case studies which, incidentally, all have harder to detect non-conformant implementations. We want to determine if the phenomenon observed remains if we restrict the set of mutants to those that are hard to kill. Thus, we repeat the same analysis but only considering mutants that are discovered by less than 20% of the unit tests. Columns 2 and 3 of table 5.5 show total number of mutants and number of mutants that are hard to detect of each case study. Note that there are no such mutants for `Socket`. Results are shown in figure 5.2 and table 5.5.

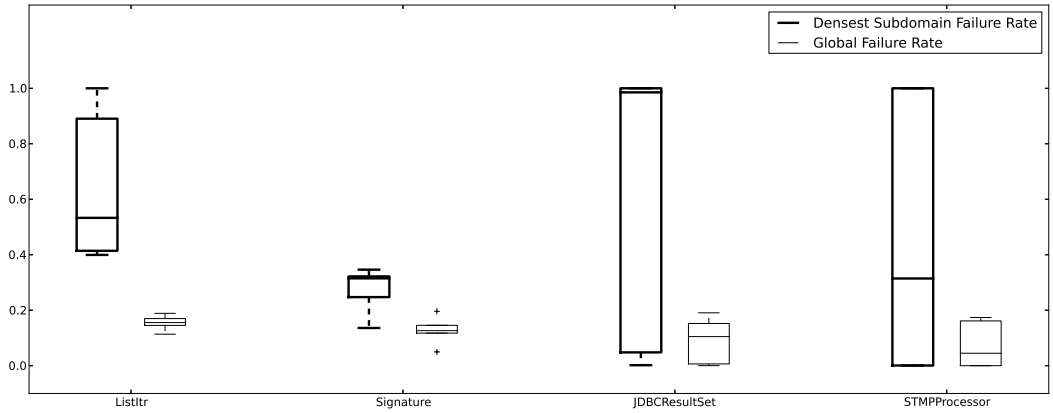


Figure 5.2: Failure rates of the densest subdomain and the entire domain of hard mutants

SUBJECT	#MUT	#HMUT	BY SUBDOMAINS					ENTIRE DOMAIN				
			Q_1	MED	Q_3	MEAN	MEAN EO	Q_1	MED	Q_3	MEAN	MEAN EO
`ListIterator`	145	14	0.41	0.53	0.56	0.62	0.62	0.14	0.16	0.16	0.16	0.16
`Signature`	96	5	0.25	0.31	0.32	0.27	0.27	0.12	0.13	0.15	0.13	0.13
`JDBCResultSet`	323	201	0.05	0.99	1.0	0.57	0.57	0.01	0.1	0.15	0.09	0.09
`SMTPProcessor`	259	176	0.0	0.31	1.0	0.49	0.49	0.0	0.01	0.16	0.08	0.08

Table 5.5: Statistical info on FR for hard mutants considering subdomains and the entire domain

As expected, when restricting the set of non-conformant implementations to those that are not easy to detect, failure rate values decrease. Still, the numbers of the subdomains derived from EPA transitions outperform, by far, those of the entire domain. In the cases of `JDBCResultSet` and `SMTPProcessor` there are revealing subdomains for at least 25% of the faulty version (in fact, in the case of `JDBCResultSet`, for 50% of them there is a subdomain with failure rate of at least 0.99). Interestingly, these are the two cases in which the number of considered implementations is relatively high. As regards failure rate mean values, as before there are no significant differences when removing outliers.

Experimental observations show that for a large number of faulty implementations there are subdomains with high failure rates. Moreover, in all case studies there are *revealing subdomains* for at least half of them. In other words, for a large number of faults partitioning the input space according to the categories defined by

transitions generates subdomains that meet the conditions under which a category partition approach results effective [JW89].

5.3 Effectiveness of Transitions

RQ.2.3. Do faults have transitions that are highly effective in detecting non-conformant implementations?

From RQ.2.2 answer one is tempted to explain the pehnomenon as follows: given a faulty implementation I, there is at least one EPA transition that, when exercised, it is very likely that I is exposed as non-conformant. Note that the existense of dense subdomains does not necessarily imply the existence of effective transitions: a transition might only exhibit a fault after being exercised several times (in which case such transition would be of low effectiveness). However, if most test cases travese the transition several times the subdomain defined by that transition would be dense in failure detection terms. What is more, a test in the subdomain for a transition t may reveal a fault while traversing a different transition. This means that the partition of inputs studied in RQ.2.2 is rather coarse grained. In this research question we aim to investigate the effectiveness of the transitions themselves. It explores if intuition on the existence of effective transitions is supported by data. This type of analysis gives us a closer look to the phenomenon observed: it is not only about determining that the inputs from some subdomain are dense, but also to observe if the non-conformance is exposed by the transition that defines the subdomain.

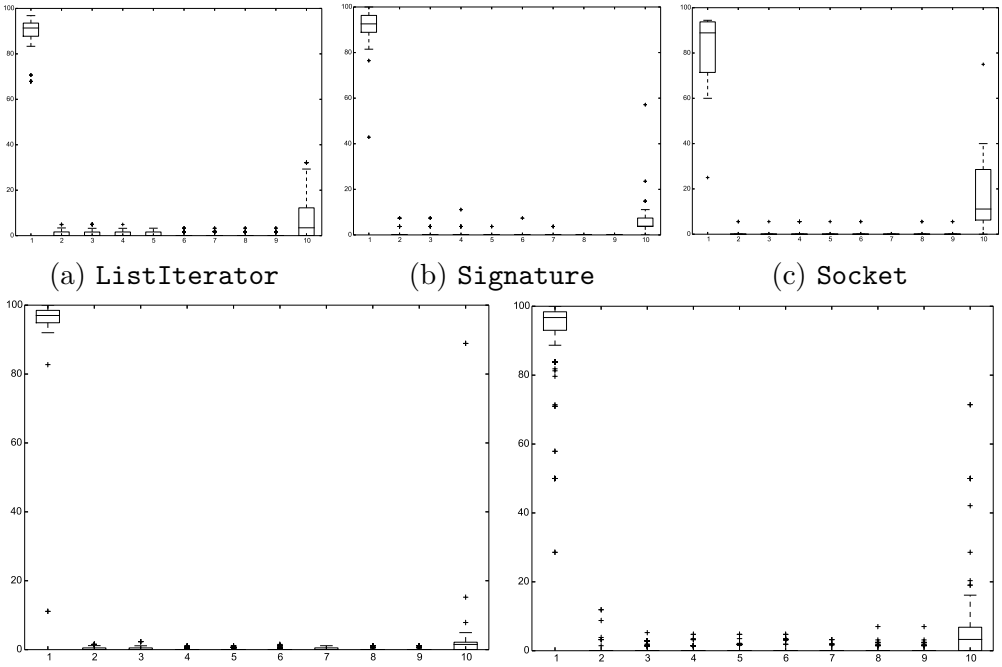
More formally, suppose that when executing unit tests over an implementation I, every time that a transition t is traversed by a unit test, the concrete execution rises an exception. That is, that every time t is traversed I is exposed as a non-conformant implementation. In such case we can consider t as highly effective for exposing I. We want to know if this is the case.

Given an implementation I and a transition t we define effectiveness of t for I (notation: $\text{eff}_{t,I}$) as the ratio between the number of times that I is exposed while traversing t and the number of times that t is traversed. It is easy to see that $\text{eff}_{t,I} \in [0, 1]$. While values near 1 denote that t is highly effective for revealing I, values near 0 indicate the opposite.

In order to understand the effectiveness of transitions for exposing non-conformant implementations we (i) divide effectiveness of transitions into ten buckets, corresponding to values in intervals $[0, 0.1], (0.1, 0.2], \ldots, (0.9, 1.0]$; and (ii) for each non-conformant implementation we count the percentage of EPA transitions whose effectiveness resides in each bucket. Boxplots of Figure 5.3 show the distribution of the effectiveness of transitions across the buckets for the five case studies.

Something similar happens in the five case studies: given a non-conformat implementation most of the transitions have low effectiveness, a few of them have high effectiveness, and in the middle there is almost nothing. The fact that in general there is at least one transition of high effectiveness supports previous observations. Suppose that a transition t is highly effective for exposing an implementation I as non-conformant -i.e., that almost always that t is traversed I raises an exception or hangs-. All the inputs of the subdomain derived from t have in common that at some point they traverse t. Therefore, it is not surprising that that subdomain results dense for I.

We observe that for non-conformant implementations in general there is a highly effective transition for it. A key question, thus, is whether there is one (or a small



(a) `ListIterator` (b) `Signature` (c) `Socket`

(d) `JDBCResultSet` (e) `SMTPProcessor`

Figure 5.3: Effectiveness of transitions for exposing non-conformant implementations

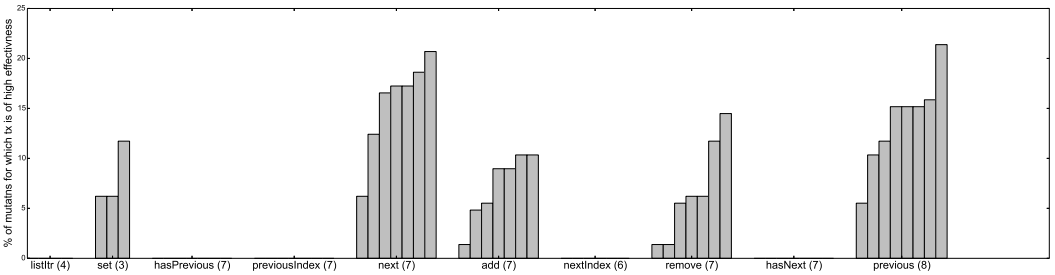
number of) highly effective transitions for all non-conformant implementations or if the highly effective transition varies according to the fault. Figure 5.4 and Figure 5.5 show the percentage of non-conformant implementations for which each transition is highly effective in each case study. There is one bar for each transition, they are grouped by actions, and for each action the number of associated transitions is indicated.

Interestingly, the transition that is highly effective for detecting a non-conformant implementation varies significantly. In the case of `ListIterator`, `Signature` and `SMTPProcessor` no transition is of high effectiveness for more than the 22% of the non-conformant implementations. In the case of `JDBCResultSet` and `Socket`, no transition is of high effectiveness for more than the 35% and 39% of the faulty implementations respectively. In general, different transitions of the same method are not higly effective for the same number of non-conformant implementation, and therefore they are not effective for the same set of implementations.

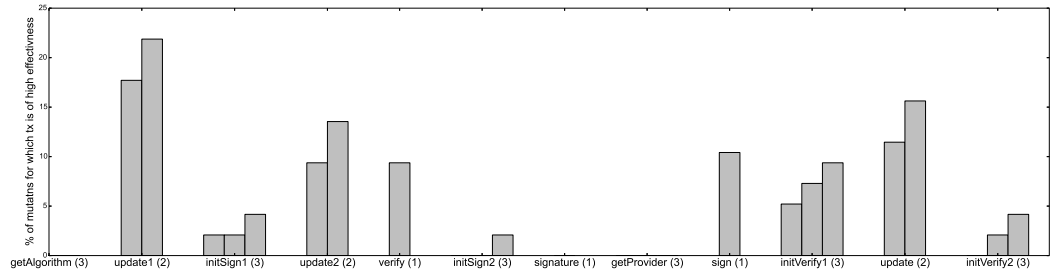
All in all, given a faulty implementation I, in general there is a transition t such that most of the times t is traveresed, I is revealed as non conformant. Additionaly, the highly effective transition vary across different non-conformat implementations. These observations reinforce the proposed criterion. For a given faulty implementation it is unknown which transitions are likely to expose it. The higher the adequacy level of a test suite, the higher the probablity of covering the highly effective transition. What is more, by adopting a category partition approch -which implies selecting some from each subdomain-, the highly effective ones are covered.

5.4 Summary of Results

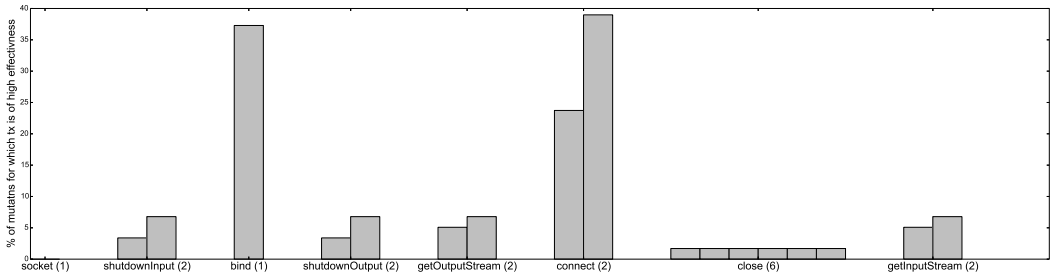
Results for RQ.2 aiming at getting a deeper understanding of the observed phenomena reinforce the results of RQ.1. Results indicate that the proposed criterion is a good predictor of structural coverage criteria (statement and branch coverage)



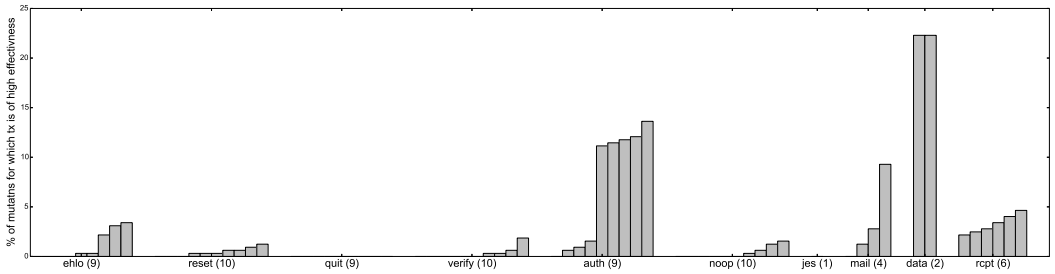
(a) `ListIterator`



(b) `Signature`



(c) `Socket`



(d) `SMTPProcessor`

Figure 5.4: Percentage of mutants for which transitions are highly effective

when applied over code under test (RQ.2.1). Correlation of EPA transition coverage and branch coverage are all 0.7 or above except for one case study which yielded 0.67. We also show that for fixed-size, test suites with the highest behavioral adequacy are statistically better in terms of structural coverage (RQ.2.1) however, as expected, the impact of EPA coverage diminishes as size increases. These results suggest that a first and early shot at producing high code coverage test suites (which could then be extended if necessary when code is available) can be achieved through EPA transition coverage.

Results also indicate that the domain partition implicitly derived from EPA transitions is likely to produce subdomains that are dense in failures, i.e., a subdomain with high failure rate (RQ.2.2). This is close to what is known to be optimal in the context of partition testing [JW89]. More specifically, in all case studies for

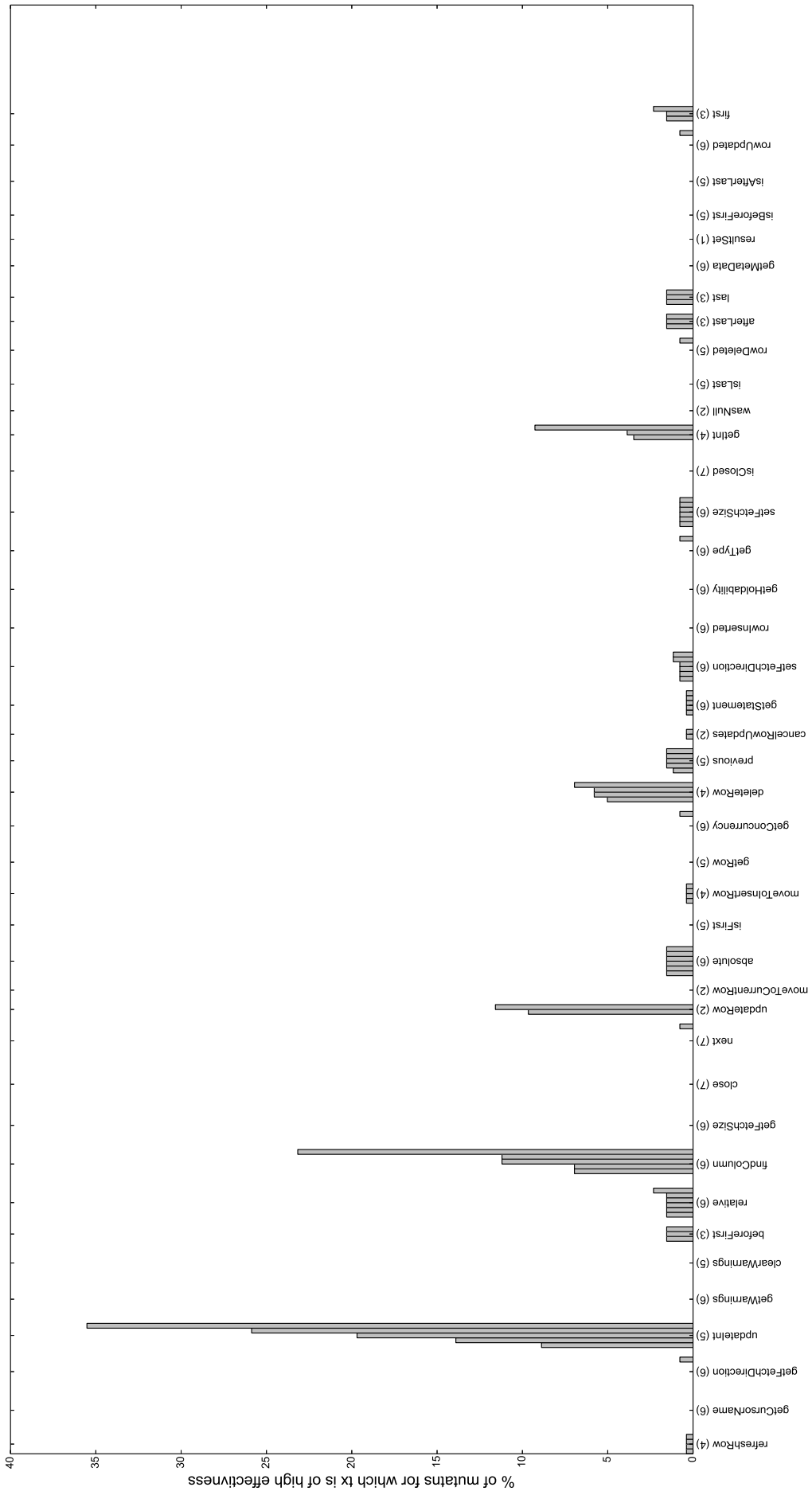


Figure 5.5: Percentage of mutants for which transitions are highly effective in `JDBCResultSet`

at least 50% of the faults, partitioning subdomains according to EPA transitions produces a 100% revealing subdomain, and that for 75% of the faults, partitioning subdomains results in at least a six-fold failure rate (i.e. density) improvement over the entire domain. Average (eliminating outliers) density per best subdomain is also high ($> 67\%$) and significantly better than for the full domain. Looking only at the hardest faults, for 50% of the faults, subdomains start from 31% revealing and are an at least two-fold (but up to two orders of magnitud) improvement over the complete domain. The Q_1 percentile and average without outliers also shows a significant improvement.

A finer grained study (RQ.2.3), investigating effectiveness of transitions (the likelihood of revealing a mutant when traversing the transition) rather than the subdomains defined by them reveals that for each fault there is almost always one transition that is highly effective in detecting it ($> 90\%$ effectiveness) while nearly all the rest of the transitions have poor effectiveness ($< 10\%$). Furthermore, we show that the effective transition is not always the same one, it depends on the fault. This is in line with previous results: as faults are a-priori unknown, it makes sense to consider more adequate those tests that cover more transitions. Test suites with highest adequacy cover all transitions, and consequently the one highly effective is exercised. However, a more significant implication is that, as uniqueness of effective subdomain per fault does not hold but does for transitions, there is an opportunity to improve on EPA transition coverage criterion.

CHAPTER 6

Comparing Transitions and Actions

Each transition of an EPA is associated with a particular action in a many-to-one relation. For instance, in the EPA of the JDK 1.4 `Socket` implementation shown in Figure 3.3 there are two different transitions corresponding to the method `getOutputStream`. The difference between those states is that while in S_3 it is legal to invoke `getInputStream`, that invocation in S_4 would result in a protocol violation. Still, `getOutputStream` is enabled on both. It is easy to see that as coverage metrics transitions properly covers [FW93b] actions. The purpose of the main research question of this chapter is to understand the improvement of covering transitions over actions, if any.

RQ.3. How do results of previous questions compare when Action Coverage is considered instead of EPA Transition Coverage?

6.1 Covering Actions

RQ.3.1 How do Action Coverage Criterion correlation values for fault detection and structural coverage compare to those of EPA Transition Coverage?

We now analize the effect of covering actions and compare the results with those of covering transitions. Table 6.1 shows Spearman's rank correlation coefficient values for both actions and transitions with fault detection and structural coverage.

Subject	Falut Detection		Statement Coverage		Branch Coverage	
	Transition	Action	Transition	Action	Transition	Action
`ListIterator`	**0.84**	0.48	0.66	**1.00**	**0.70**	0.58
`Signature`	**0.73**	0.68	0.63	**0.79**	0.70	**0.75**
`Socket`	0.72	**0.76**	0.77	**0.93**	0.77	**0.89**
`JDBCResultSet`	0.68	**0.77**	**0.67**	0.60	**0.66**	0.57
`SMTPProcessor`	**0.69**	0.55	0.70	**0.92**	0.73	**0.76**

Table 6.1: Correlation between transitions and actions with fault detection and structural coverage

As regards fault detection, covering transitions perform better than covering actions in three out of five case studies. Additionaly, the correlation values for transition coverage are more stable than those of action coverage.

In terms of structural coverage, action coverage perform better than transition coverage in seven out of ten cases. This is not surprising because correlation measures how well the relationship between two variables can be described by a monotonic function. In the case of action and structural coverage, it measure to what extent is true that as more actions are executed more code is exercised. Every time that an action that had not been executed before is executed, some portion of the code that had not been exercised before is exercised. This observation do not hold for transition coverage, since covering two different transitions of the same action may not necessarily imply covering a new portion of the code.

The case of `ListIterator` is particulary interesting. As already mentioned, it has a quite rich protocol and a extremly simple code structure. The correlation between action coverage and statement coverage is the best possible, whereas the one with fault detection is the lowest one. On the one hand, the poor performance of actions regarding fault detection indicates that just executing different actions is not enough for killing more mutants. On the other hand, covering more transitions (which implies executing actions from different enabledness states) has a possitive impact on the number of mutants killed.

6.2 Actions vs. Transitions From a Category Partition Perspective

RQ.3.2 How do failure rates of subdomains defined by Action Coverage compare to those of subdomains defined by Transition Coverage?

We now analize the failure rates of the subdomains derived from covering actions in the same way as we did with transitions. We consider that a unit test u belongs to the subdomain associated to an action a if and only if a is invoked in u. Note that, as with transitions, the partition may result in overlapping subdomains. For instance, the unit test of Listing 2.2 belongs to the subdomains associated to both actions *push* and *pop*. Figure 6.1 shows boxplots of failure rates for densest subdomains for both transitions and actions. Numerical values of percentils, means and means excluding outliers are shown in table 6.2.

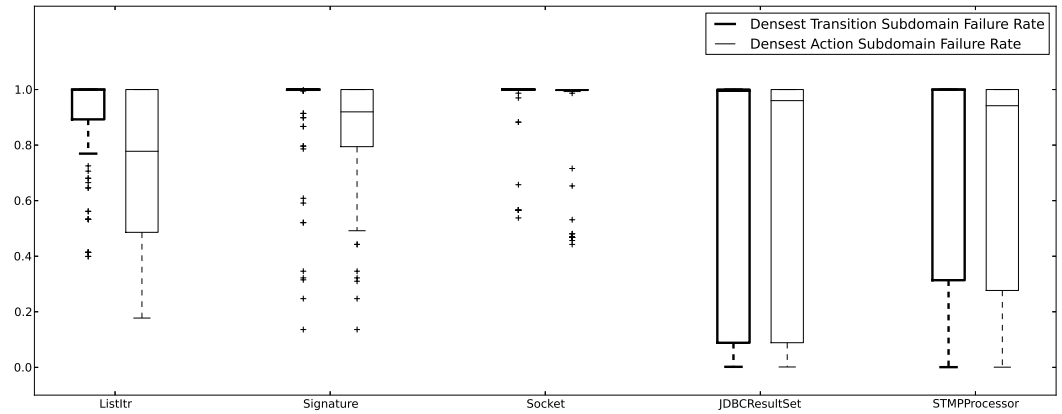


Figure 6.1: Failure rates of the densest subdomains defined by transitions and actions

Subject	By Transition Subdomains					By Action Subdomains				
	Q_1	Med	Q_3	Mean	Mean EO	Q_1	Med	Q_3	Mean	Mean EO
`ListIterator`	0.89	1.0	1.0	0.91	0.95	0.49	0.78	1.0	0.71	0.71
`Signature`	1.0	1.0	1.0	0.93	0.98	0.79	0.92	1.0	0.87	0.92
`Socket`	1.0	1.0	1.0	0.94	0.99	0.99	1.0	1.0	0.89	0.96
`JDBCResultSet`	0.09	1.0	1.0	0.67	0.67	0.09	0.96	1.0	0.65	0.65
`SMTPProcessor`	0.31	1.0	1.0	0.71	0.71	0.28	0.94	1.0	0.65	0.65

Table 6.2: Statistical info of failure rates considering subdomains derived from transitions and actions

Whereas in the case of transitions there is a revealing subdomain for at least 50% of non-conformant implementations, in the case of actions this observation holds for only 25% of them. Still, failure rates values of actions' subdomains are acceptable: subdomains' failure rates for 50% of the mutants range (at least) between 0.78 and 1.00 depending on the case study.

Also in this case we repeat the analysis restricting the universe of mutants to those that are hard to detect (i.e., that are killed by less than 20% of the unit tests). Boxplots are shown in figure 6.2 and numerical values in table 6.3. Except for the case of `ListIterator`, where clearly failure rate values of transitions' subdomains outperform those of actions' subdomains, results for actions and transitions do not present major differences.

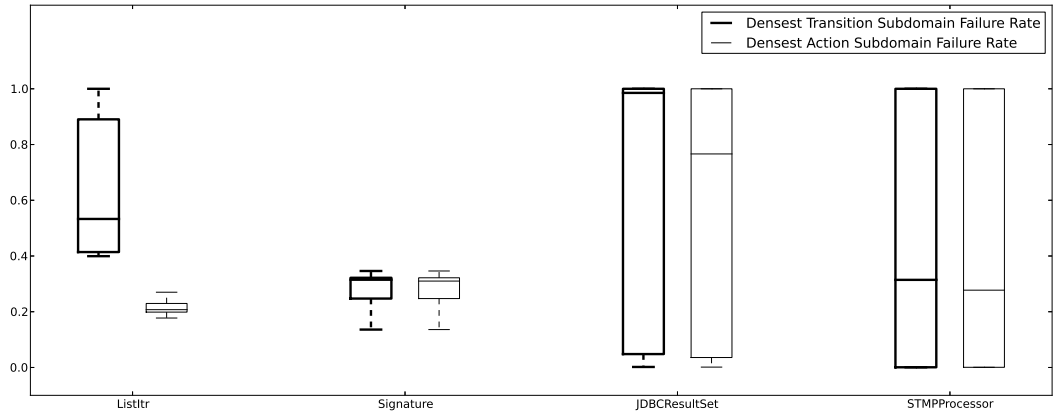


Figure 6.2: Failure rates of the densest subdomains defined by transitions and actions for hard mutants

Subject	#Mut	#HMut	By Trasition Subdomains					By Action Subdomains				
			Q_1	Med	Q_3	Mean	Mean EO	Q_1	Med	Q_3	Mean	Mean EO
`ListIterator`	145	14	0.41	0.53	0.56	0.62	0.62	0.2	0.21	0.21	0.21	0.21
`Signature`	96	5	0.25	0.31	0.32	0.27	0.27	0.25	0.31	0.32	0.27	0.27
`JDBCResultSet`	323	201	0.05	0.99	1.0	0.57	0.57	0.04	0.77	1.0	0.55	0.55
`SMTPProcessor`	259	176	0.0	0.31	1.0	0.49	0.49	0.0	0.28	1.0	0.46	0.46

Table 6.3: Statistical info on FR for hard mutants considering transition and action subdomains

The partition of inputs as studied in this section is, however, rather coarse grained for comparing actions and transitions. Consider, for instance, the unit test of figure 2.2. A faulty implementation I may fail when *push* is invoked, but since the input belongs to the subdomains derived from both actions (*push* and *pop*), the killing of I is count for both, although the *pop* statement is not even reached when executing the unit test on implementation I. A finer grained perspective for evaluating differences between transitions and actions is developed in the following section.

6.3 Effectiveness of Actions vs. Effectiveness of Transitions

RQ.3.3 How do effectiveness of actions compare to the effectiveness of transitions for revealing faults?

As with transitions, we define the effectiveness of an action a for an implementation I (notation $\text{eff}_{a,I}$) as the ratio of the number of times that I is exposed as a non-conformant implementation when executing a and the total number of times that a is executed on I.

Note that it is not possible to have an action with greater effectiveness to all its associated transitions. Note also that there are some situations in which transitions cannot improve the effectiveness of actions. There are actions associated with only one transition. While this situation is not present neither in `SMTPProcessor` nor in `ListIterator`, this is the case of 5 out of 14 actions of `Signature`, 1 out of 41 actions of `JDBCResultSet` and 1 out of 7 actions of `Socket`. When the most effective transition is the only transition of some action, there is no distiction between their highest effectiveness. In all such cases it is meaningless to study the distinction between transitions and actions. Besides, there are some faulty implementations such that there is an action a whose effectiveness equals 1 -i.e., every time a is executed the implementation is revealed as non-conformant. Again, no transition can improve actions in this case. Still, there are many implementations for which the described situations do not hold, and therefore transitions may improve the effectiveness of actions. Table 6.4 shows information regarding improvable and not improvable faulty implementations. The second column of the table specifies the number of mutations that resulted in non-conformant implementations; the third and fourth columns show the number of faulty implementation for which there is an action with effectiveness 1 and for which the most effective transition is the only transition of that action; the fifth and sixth columns expose the number of faulty implementation for which transitions cannot and can improve actions respectively. Note that column 5 do not always correspond to the sum of columns 3 and 4, since there are implementations for which there is an action a of effectiveness 1 and a has only one associated transition. For instance, action `bind` of `Socket` has effectiveness 1 for 15 faulty implementations, and it has only one associated trasition (see figure 3.3).

Subject	#Total	#$\text{eff}_{a,I} = 1$	#1a-1t	#Filtered	#Improvable
`ListIterator`	145	45 (31.03%)	0 (0%)	45 (31.03%)	100 (68.97%)
`Signature`	96	36 (37.5%)	26 (27.08%)	43 (44.79%)	53 (55.21%)
`Socket`	59	44 (74.58%)	23 (38.98%)	52 (88.14%)	7 (11.86%)
`JDBCResultSet`	259	116 (44.79%)	0 (0%)	116 (44.79%)	143 (55.21%)
`SMTPProcessor`	317	206 (64.98%)	0 (0%)	206 (64.98%)	111 (35.02%)

Table 6.4: Improvable and not improvable non-conformant implementations

We now analize what happens when there is a chance for improvement. For each faulty implementation such that transitions could improve actions we calculate the highest effectiveness of both, transitions and actions. Boxplots of Figure 6.3 shows the percentage of improvement of transitions over actions for the five case studies. Numerical values of Q_1, median, Q_3, mean and mean excluding outliers are shown in table 6.5.

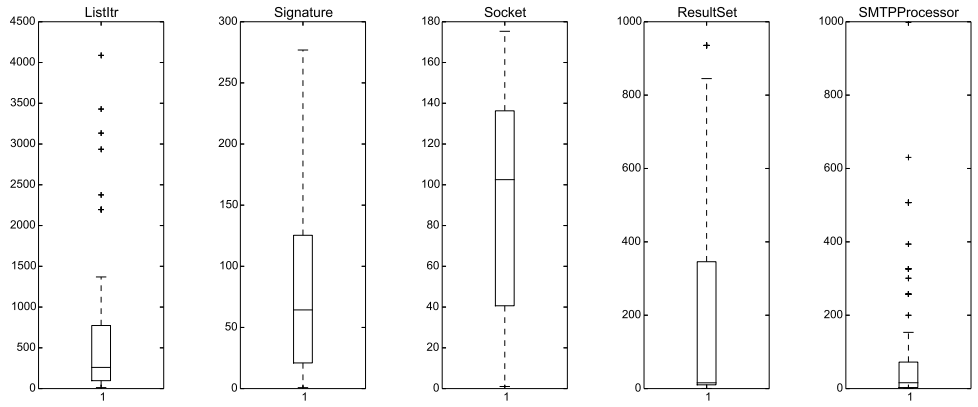


Figure 6.3: Improvements of transitions over actions

Subject	Q_1	Median	Q_3	Mean	Mean EO
`ListIterator`	97.25%	260.23%	773.77%	708.52%	350.49%
`Signature`	20.98%	64.34%	125.37%	81.32%	81.32%
`Socket`	32.96%	102.52%	103.43%	90.39%	90.39%
`JDBCResultSet`	10.27%	15.73%	345.83%	193.78%	163.51%
`SMTPProcessor`	2.02%	6.27%	128.16%	110.10%	54.76%

Table 6.5: Numerical values of improvement of transitions over actions

The level of improvement of transitions over actions vary significantly across case studies. Nevertheless, the improvement is considerably high in all cases for a large number of non-conformant implementation. Mean values of improvents range between 81.32% and 708.52%, and when excluding outliers between 54.76% and 350%. From this we can conclude that transitions are much more effective than actions. This observation reinforce the idea of covering transitions (instead of just executing actions) for revealing non conformant implementatinos.

6.4 Summary of Results

Experimentation regarding RQ.3, aimed at investigating that promising results for EPA transition coverage are not simply because the criterion is a refinement of a action coverage, provides positive indicators. Results suggest that for correlation with fault detection, in most cases EPA transition coverage performs better than action coverage.

Results also indicate that subdomains defined by actions are less dense than those derived from transitions. The most interesting results arise when comparing effectiveness of individual transitions and actions; here the former significantly improves the latter. The average after eliminating outliers shows improvements starting at 50% and up to 350%; while looking at percentiles, for 50% (resp. 75%) of faults transition effectiveness improves over action effectiveness in 4 out of 5 case studies between 15% and 260% (resp 10% and 97%). The last case study shows a smaller improvement 6% (resp. 2%). These results suggest that EPA transition coverage can provide benefits over action coverage, and justify studying a combined criterion.

CHAPTER 7

Combining Transitions and Actions

Results of Chapter 6 indicate that there may be an opportunity at combining Action and EPA Transition Coverages in order to define the level of adequacy of a test suite. In this chapter we define and study such adequacy criterion in order to answer the fourth main research question.

RQ.4. How do results for RQ.1 and RQ.2 compare when a combined Action-EPA Transition Coverage criterion is adopted?

7.1 A Combined Criterion

The combined criterion involves covering both, actions and transitions. The rationale behind the criterion is that if an enabled action has not been executed, then choose to execute it. Covering a new action improves code coverage and additionaly also implies to exercie a new transition. When all available actions have already been executed then choose one such that its execution covers a yet uncovered transition. This way, strengths of both transtions and actions are being exploited. We define the adequacy of a test suite according to the combined criterion as follows.

Definition 7 (Action-Transition Adequacy Level)**.** *Let L be a protocol LTS, A its EPA and TS a test suite. The action-transition adequacy level of TS is defined as the ordered pair (j, k) where j indicates the percentage of invoked actions and k the percentage of transitions of A that are covered by the α-abstracted execution of the unit tests of TS.*

Two values are involved in this definition: number of actions invoked by a test suite and number of transitions that are exercised by the α-abstracted execution of the test suite. Since we want to calculate how this criterion correlates with fault detection and structural coverage, given a set of test suites we need to determine an order between them according to the above definition, which we do in a lexicographical manner.

Definition 8 (Action-Transition Adequacy Order)**.** *Let L be a protocol LTS, A its EPA, and TS_1 and TS_2 two test suites with adequacy level (j_1, k_1) and (j_2, k_2) respectively. If $j_1 > j_2$ or $j_1 = j_2$ and $k_1 > k_2$ then TS_1 has greater level of adequacy*

than TS_2. If $j_1 = j_2$ and $k_1 = k_2$ both are equally adequate. Otherwise, TS_2 has greater level of adequacy than TS_1.

7.2 The Criterion as Fault Detection and Structural Coverage Predictor

RQ.4.1 How do the combined criterion correlation values for fault detection and structural coverage compare to those of EPA Transition Coverage and Action Coverage?

Spearman's correlation values for fault detection and structural coverage are shown in table 7.1 for the three criteria: action coverage (A), transition coverage (T) and the combined coverage criterion ((A+T)).

Subject	Fault Detection			Statement Coverage			Branch Coverage		
	A+T	T	A	A+T	T	A	A+T	T	A
`ListIterator`	**0.84**	**0.84**	0.48	0.71	0.66	**1.00**	0.50	**0.70**	0.58
`Signature`	**0.73**	**0.73**	0.68	0.78	0.63	**0.79**	**0.79**	0.70	0.75
`Socket`	0.75	0.72	**0.76**	0.74	0.77	**0.93**	0.70	0.77	**0.89**
`JDBCResultSet`	**0.80**	0.68	0.77	**0.78**	0.67	0.60	**0.91**	0.66	0.57
`SMTPProcessor`	**0.69**	**0.69**	0.55	0.69	0.70	**0.92**	0.66	0.73	**0.76**

Table 7.1: Correlation between the combined criterion ,transitions and actions with fault detection and structural coverage

As regards fault detection, in four out of five case studies the combined criterion outperforms action and transition criteria. For the remaining case study (`Socket`), the correlation for the combined criterion is just 0.01 less than the best one for this subject. With respect to statement coverage, action coverage is the best in four out of five case studies. As mentioned before, this is not surprising considering that always that a yet unexecuted action is executed some yet unexcersised statements are excercised (some of those corresponding to the action). Still, values corresponding to the combined criterion are high and more stable than those of the other criteria. All of them range between 0.69 and 0.78. More interestengly, in the case of branch coverage in three cases criteria involving covering transitions have higher correlation values than those of action coverage. Executing different branches is often related to the internal state of objects. The existance of different transitions of the same action in the EPA also reflects differences on internal object states. Therefore, covering more transitions implies executing actions from distinct internal states, which in turn increases the chances of covering more branches.

7.3 Hypothesis Tests for Fixed-Size Test Suites

RQ.4.2. Given a fixed test suite size, do test suites with higher Adequacy Level according to the combined criterion perform better in terms of fault detection and structural coverage than those with lower adequacy level?

As explained in Chapter 4, the test suite size impacts on the effectiveness of the test suite. As with previously studied criteria, those tests that achieve higher

coverage are also longer, so we again analyse the results on a "by size" basis. That is, given a size we aim to study if the test suites with higher Action-Transition Adequacy Level are likely to detect more faults than those with lower adequacy when fixing the size of test suites.

For performing the analysis we use the same set of bins of Chapter 4 and Chapter 5. Recall that we divide the test suites in ten bins, and that the difference of size between test suites of the same bin do not exceed 10% of the difference of size between the largest and the smallest overall test suites. We want to see whether picking a test suite from the best test suites according to the combined criterions is more likely to:

1. give a test suite with higher fault detection ability than picking it from the rest of the test suites of that bin; and

2. give a test suite which has higher structural coverage than picking it from the rest of the test suites of that bin.

In the analysis performed on the EPA-Transition adequacy criterion we considered that the "best" test suites were those that achieved at least the 80% of the coverage achieved by the test suite with highest coverage of that bin. Since in the combined Action-Transition Criterion the level of adequacy of a test suite is given by an ordered pair, we cannot repeat the same strategy for dividing the test suites of a bin. In this case we just order the test suites according to Definition 8 and consider the first 20% of them as the test suites with high coverage and the remaining 80% as those with low coverage. Note that dividing them this way may leave two test suites with identical coverage in different groups. That is, one of them in the group of test suites with high coverage and the other test suite in the group of test suites with low coverage. The following sequence of coverages illustrates this situation:

$$\text{All} = [(70,80),(60,80),(60,80),(50,50),(50,40),(50,40),(50,20),(40,10),(30,10),(30,10)]$$
$$\text{High} = [(70,80),(60,80)]$$
$$\text{Low} = [(60,80),(50,50),(50,40),(50,40),(50,20),(40,10),(30,10),(30,10)]$$

In order to avoid such situation we shift all *tied* test suites to the high side. In the example, both test suite with coverage $(60,80)$ would be included in the group of high coverage.

As explained in Chapter 4 and Chapter 5, neither detected fault data nor structural coverage data fit standard distributions. Thus, as before, we again use the Mann-Whitney U test. Again, the assumptions of this hypothesis test are met: test suites with high and low coverage form disjoint sets, and values cosidered are ordinal measurements. Under the null hypothesis the probability of a random observation from one population P_1 exceeding a random observation from the second population P_2 equals the probability of an observation from P_2 exceeding an observation from P_1. Under the alternative hypothesis the probability is not equal to 0.5. We again reject the null hypothesis when the ρ-value resulting from the hypothesis test is less than 0.05. In this case, rejecting the null hypothesis means that tests with high Action-Transition Coverage are better in terms of fault detection and/or code coverage than those of low coverage.

We again evaluate the magnitude of the improvement by using the Vargha and Delaney's A_{12} effect size (ES). In this case, in a given bin A_{12} estimates the probability that choosing a tests suite of high Action-Transition coverage detects more faults or achieve higher structural coverage than one chosen randomly from the pop-

ulation of low coverage. We again report the confidence interval (CI) for the effect size stated at the 95% confidence level.

Table 7.2 shows the test results for all subjects. Each row of the table corresponds to one bin. The second column specifies the test suite size interval of each bin. The third indicates the threshold ordered pair of Action-Transition Coverage that a test suite must achieve to be considered adequate (i.e., of high coverage). The fourth and fifth columns show the number tests suites that are adequate and not adequate respectively. Columns six to eiqht exhibit the ρ-value of the Mann-Whitney test, the A_{12} effect size and its confidence interval regarding fault detection. Columns nine to eleven and twelve to fourteen show these values for statement and branch coverage respectively.

Results indicate that Action-Transition Coverage makes a difference in terms of fault detection for tests suites of the same size. In 44 out of 50 bins (88% of the bins) there is statistical evidence that test suite with higher coverage of the combined criterion perform better than those of lower coverage. These results are better than those of the EPA-Transition Coverage Criterion, in which the statistical evidence was present in 78% of the bins. Again, those bins on which there is no statistical evidence correspond to bins containing very large test suites.

Results regarding Statement Coverage are quite interesting. On the one hand, in the cases of `Signature`, `Socket`, `SMTPProcessor` and `JDBCResultSet` there is statistical evidence of the convinience of high coverage test suites in all bins. On the other hand, there is no evidence for bins of `ListIterator`. As already explained, its code is quite simple and therefore it is easy to achieve high statement coverage. As with the EPA-Transition Criterion, the Action-Transition Criterion do not imply an improvement in this case. Considering the bins of all case studies toghether, in 80% of them there is statistical evidence, which is also an improvement with respect to the 68% of the EPA-Transition Criterion. Finally, as regards Branch Coverage results show that there is statistical evidence in 48 out of 50 bins (96%), which also improves the 86% of the EPA-Transition Criterion.

7.4 Summary of Results

The set of results corresponding to RQ.4 shows that a coverage criterion based on actions and transitions outperforms criteria of those of actions and transitions (when considered in isolation) as a fault detection predictor in all the case studies except for one, in which the highest correlation is only 0.01 greater. Values for the combined criterion range between 0.69 and 0.84. In terms of structural coverage, correlation values of Action Coverage are better than those of the combined criteria. Nevertheless, values of Action-Transition Coverage are still high and stable across all the five case studies. Regarding branch coverage, none of the criteria considered shows great advantages over the rest.

Also, when considering fixed-size test suites we see that test with high level of adequacy according to the Action-Transition Coverage Criterion are better than those of low coverage regarding both, fault detection and strucutral coverage. We find statistical evidence in a larger number of bins than for the Transition Coverage Criterion.

In conclusion, we believe that combining actions and transitions exploits the benefits of Action Coverage and Transition Covergate in a single criterion.

Subject	Action-Transition Coverage				Fault Detection			Statement Coverage			Branch Coverage		
	Size	Thr	$\#TS_{\geqslant Thr}$	$\#TS_{<Thr}$	p	ES	CI	p	ES	CI	p	ES	CI
ListIterator	[50,99]	(10,31)	65	247	~ 0	0.79	[0.73,0.85]	$0.18^{\dagger}$	0.56	[0.53,0.61]	~ 0	0.78	[0.73,0.84]
	[100,149]	(10,36)	67	220	~ 0	0.81	[0.75,0.86]	$0.53^{\dagger}$	0.53	[0.51,0.55]	~ 0	0.79	[0.75,0.82]
	[150,199]	(10,39)	65	216	~ 0	0.72	[0.66,0.79]	$0.77^{\dagger}$	0.51	[0.51,0.52]	~ 0	0.7	[0.67,0.74]
	[200,249]	(10,42)	66	232	~ 0	0.75	[0.70,0.81]	$0.90^{\dagger}$	0.51	[0.51,0.51]	~ 0	0.58	[0.54,0.61]
	[250,299]	(10,44)	65	237	~ 0	0.77	[0.71,0.83]	$0.82^{\dagger}$	0.51	[0.50,0.52]	~ 0	0.58	[0.55,0.61]
	[300,349]	(10,45)	71	221	~ 0	0.65	[0.58,0.72]	$0.45^{\dagger}$	0.50	[0.50,0.50]	~ 0	0.57	[0.55,0.59]
	[350,399]	(10,46)	72	169	~ 0	0.64	[0.57,0.71]	$0.84^{\dagger}$	0.53	[0.51,0.54]	0.01	0.54	[0.52,0.56]
	[400,449]	(10,47)	56	131	$0.06^{\dagger}$	0.58	[0.50,0.66]	$0.97^{\dagger}$	0.51	[0.51,0.51]	0.01	0.55	[0.53,0.58]
	[450,499]	(10,48)	36	137	0.03	0.61	[0.52,0.70]	$1.00^{\dagger}$	0.50	[0.50,0.51]	$0.05^{\dagger}$	0.54	[0.52,0.56]
	[500,549]	(10,48)	32	95	$0.18^{\dagger}$	0.55	[0.46,0.65]	$1.00^{\dagger}$	0.56	[0.55,0.57]	$0.07^{\dagger}$	0.54	[0.51,0.56]
Signature	[50,99]	(11,19)	59	228	~ 0	0.88	[0.80,0.96]	~ 0	0.91	[0.84,0.99]	~ 0	0.83	[0.74,0.91]
	[100,149]	(12,18)	46	178	~ 0	0.88	[0.78,0.98]	~ 0	0.92	[0.86,0.98]	~ 0	0.84	[0.72,0.96]
	[150,199]	(12,20)	51	189	~ 0	0.86	[0.77,0.96]	~ 0	0.87	[0.82,0.93]	~ 0	0.82	[0.71,0.92]
	[200,249]	(12,23)	53	191	~ 0	0.87	[0.78,0.96]	~ 0	0.89	[0.82,0.96]	~ 0	0.84	[0.74,0.93]
	[250,299]	(12,25)	54	190	~ 0	0.88	[0.81,0.96]	~ 0	0.81	[0.75,0.87]	~ 0	0.84	[0.78,0.91]
	[300,349]	(12,25)	56	188	~ 0	0.91	[0.83,1.00]	~ 0	0.79	[0.70,0.88]	~ 0	0.74	[0.66,0.83]
	[350,399]	(12,26)	65	250	~ 0	0.74	[0.64,0.83]	~ 0	0.80	[0.75,0.85]	~ 0	0.73	[0.68,0.79]
	[400,449]	(12,26)	58	182	~ 0	0.71	[0.60,0.82]	~ 0	0.75	[0.70,0.80]	~ 0	0.71	[0.63,0.79]
	[450,499]	(12,27)	51	200	~ 0	0.84	[0.75,0.93]	~ 0	0.73	[0.65,0.81]	0.02	0.63	[0.58,0.68]
	[500,549]	(12,26)	49	162	~ 0	0.8	[0.69,0.90]	~ 0	0.83	[0.78,0.88]	~ 0	0.68	[0.60,0.76]
Socket	[0,12]	(7,7)	43	127	~ 0	0.71	[0.64,0.77]	~ 0	0.89	[0.81,0.96]	~ 0	0.87	[0.80,0.94]
	[13,25]	(8,9)	85	212	~ 0	0.73	[0.67,0.78]	~ 0	0.95	[0.91,0.99]	~ 0	0.89	[0.85,0.93]
	[26,38]	(8,12)	61	230	~ 0	0.71	[0.65,0.78]	~ 0	0.81	[0.75,0.86]	~ 0	0.77	[0.71,0.83]
	[39,51]	(8,13)	74	226	~ 0	0.6	[0.55,0.66]	~ 0	0.71	[0.66,0.76]	~ 0	0.7	[0.64,0.76]
	[52,64]	(8,14)	74	244	0.01	0.56	[0.52,0.60]	~ 0	0.61	[0.56,0.66]	~ 0	0.61	[0.56,0.67]
	[65,77]	(8,14)	130	167	0.01	0.54	[0.51,0.57]	~ 0	0.61	[0.57,0.65]	~ 0	0.63	[0.58,0.67]
	[78,90]	(8,15)	80	233	0.04	0.52	[0.51,0.54]	~ 0	0.59	[0.56,0.63]	~ 0	0.62	[0.58,0.66]
	[91,103]	(8,15)	94	190	$0.06^{\dagger}$	0.52	[0.50,0.53]	~ 0	0.57	[0.54,0.60]	~ 0	0.59	[0.56,0.63]
	[104,116]	(8,15)	69	108	$0.15^{\dagger}$	0.51	[0.49,0.54]	0.02	0.55	[0.51,0.59]	0.04	0.55	[0.51,0.59]
	[117,129]	(8,15)	12	41	0.04	0.55	[0.53,0.57]	~ 0	0.58	[0.55,0.61]	~ 0	0.60	[0.56,0.64]
SMTPProcessor	[0,149]	(9,19)	42	154	~ 0	0.78	[0.69,0.87]	~ 0	0.85	[0.78,0.92]	~ 0	0.82	[0.74,0.90]
	[150,299]	(10,23)	57	206	~ 0	0.87	[0.81,0.93]	~ 0	0.92	[0.86,0.97]	~ 0	0.9	[0.84,0.96]
	[300,449]	(10,31)	58	189	~ 0	0.79	[0.71,0.86]	~ 0	0.83	[0.77,0.90]	~ 0	0.83	[0.76,0.90]
	[450,599]	(10,35)	55	204	~ 0	0.73	[0.65,0.81]	~ 0	0.7	[0.61,0.78]	~ 0	0.72	[0.64,0.80]
	[600,749]	(10,37)	53	201	0.04	0.61	[0.51,0.70]	0.01	0.64	[0.55,0.73]	0.01	0.63	[0.54,0.72]
	[750,899]	(10,40)	57	193	0.01	0.63	[0.54,0.73]	~ 0	0.7	[0.62,0.78]	~ 0	0.7	[0.61,0.78]
	[900,1049]	(10,40)	72	210	0.04	0.59	[0.51,0.68]	0.02	0.6	[0.52,0.69]	0.02	0.6	[0.52,0.69]
	[1050,1199]	(10,43)	56	212	$0.09^{\dagger}$	0.59	[0.47,0.70]	0.03	0.61	[0.52,0.71]	0.01	0.66	[0.55,0.77]
	[1200,1349]	(10,43)	61	197	0.02	0.62	[0.53,0.71]	~ 0	0.64	[0.56,0.72]	0.01	0.63	[0.54,0.71]
	[1350,1499]	(10,44)	53	170	$0.09^{\dagger}$	0.57	[0.49,0.66]	~ 0	0.63	[0.56,0.71]	~ 0	0.65	[0.56,0.73]
JDBCResultSet	[200,399]	(38,97)	32	124	~ 0	0.82	[0.74,0.90]	~ 0	0.93	[0.88,0.99]	~ 0	0.78	[0.69,0.87]
	[400,599]	(39,104)	56	213	~ 0	0.72	[0.64,0.79]	~ 0	0.89	[0.84,0.94]	~ 0	0.72	[0.65,0.80]
	[600,799]	(40,100)	58	225	~ 0	0.8	[0.73,0.87]	~ 0	0.95	[0.91,0.99]	~ 0	0.75	[0.68,0.83]
	[800,999]	(40,116)	55	208	~ 0	0.78	[0.70,0.85]	~ 0	0.9	[0.84,0.95]	~ 0	0.77	[0.70,0.85]
	[1000,1199]	(40,126)	50	185	~ 0	0.81	[0.74,0.88]	~ 0	0.9	[0.85,0.96]	~ 0	0.83	[0.77,0.90]
	[1200,1399]	(41,118)	52	202	~ 0	0.87	[0.82,0.92]	~ 0	0.98	[0.95,1.00]	~ 0	0.86	[0.81,0.91]
	[1400,1599]	(40,139)	53	204	~ 0	0.86	[0.81,0.91]	~ 0	0.97	[0.93,1.00]	~ 0	0.86	[0.80,0.91]
	[1600,1799]	(41,130)	57	214	~ 0	0.89	[0.84,0.94]	~ 0	0.95	[0.91,0.99]	~ 0	0.88	[0.84,0.93]
	[1800,1999]	(41,135)	53	194	~ 0	0.86	[0.81,0.91]	~ 0	0.95	[0.90,0.99]	~ 0	0.87	[0.82,0.92]
	[2000,2199]	(41,139)	58	207	~ 0	0.84	[0.78,0.89]	~ 0	0.96	[0.92,1.00]	~ 0	0.88	[0.83,0.93]

Table 7.2: Mann-Whitney test results for fault detection and code coverage. Values not statistically significant are marked with †.

CHAPTER 8

Test Suite Prioritisation Analysis

This chapter addresses the last research question, which aims to understand the quality of prioritisation techniques based on EPA coverage.

> **RQ.5.** How do EPA-based coverage prioritisation techniques compare to random and code coverage ones in terms of fault detection rate?

We investigate the quality of EPA-coverage-based prioritisation techniques and compare them to techniques based on code coverage and random orders. We examine the effects of using random, EPA and code coverage criteria on the fault detection rates of the prioritised test suites. We have found that prioritisation based on EPA-coverage is always better than a random approach, and also that they tend to achive high failure-rates almost at the same speed or faster than those based on white-box code coverage.

8.1 The Test Suite Prioritisation Problem

Broadly speaking, test case prioritisation's goal is to order unit tests in such a way that faults are detected as early as possible. Having an early high rate of fault detection gives an earlier feedback on system defects. Therefore, it allows developers to locate and fix faults sooner. Besides, the feedback can also provide early evidence that some quality goals have not been met, allowing earlier strategic decisions about release schedules. Moreover, in presence of a limited testing time budget, the order of unit tests may make the difference between finding or not several bugs. This becomes particularly important when such budget is low [WSKR06].

Formally, the test suite prioritisation problem is defined as follows [EMR00]:

> Test Case Prioritisation Problem
> Given: TS, a test suite; PT the set of permutations of TS; and f, a function from PT to the real numbers.
> Problem: Find $P' \in PT$ such that $\forall P'' \in PT.f(P') \geq f(P'')$.

According to this definition, PT represents the set of all possible orderings of the unit tests of TS. When the function f is applied to any ordering it yields an award

value for that ordering. Given a permutation P, the greater the value of $f(P)$, the better. Thus, the permutation P with highest $f(P)$ is the best possible according to f.

8.2 Effectiveness Measure

In [EMR00] authors presented a metric that has become the standard to measure the efficiency of prioritisation techniques. This metric measures how quickly a test suite detects faults by calculating the weighted Average of the Percentage of Faults Detected (APFD) over the life of a test suite. The APFD value ranges from 0 to 100, and the higher the APFD, the faster the rate of fault detection. It is defined as follows:

APFD Measure
Let TS be a test suite containing n unit tests and let F be a set of m faults revealed by TS. Let TF_i be the position of the first unit test in TS which reveals fault i. The APFD for test suite ordering of TS is given by the equation:

$$\text{APFD} = 1 - \frac{TF_1+TF_2+...+TF_m}{nm} + \frac{1}{2n}$$

To see how this metric works, let us consider an example from [EMR00]. Suppose that we have a program with 10 faulty versions $f_1 \ldots f_{10}$ and a test suite TS of 5 unit tests $ut_1 \ldots ut_5$ with fault detection ability described in table 8.1 (an entry [ut_i, f_j] marked with an x indicates that unit test ut_i exposes f_j as a faulty version).

	f_1	f_2	f_3	f_4	f_5	f_6	f_7	f_8	f_9	f_{10}
ut_1	x				x					
ut_2	x				x	x	x			
ut_3	x	x	x	x	x	x	x			
ut_4					x					
ut_5								x	x	x

Table 8.1: Test suite and fault exposed by unit tests

Suppose we place the unit tests in order [ut_1-ut_2-ut_3-ut_4-ut_5] to form a prioritised test suite PTS_1. Figure 8.1 shows the percentage of detected faults versus the fraction of the test suite PTS_1 used. After running unit test ut_1, 2 of the 10 faults are detected. Thus, 20% of the faults have been detected after 0.2 of test suite PTS_1 has been used. After running unit test ut_2, 2 more faults are detected and thus 40% of the faults have been detected after 0.4 of the test suite has been used. In Figure 8.1, the area inside the inscribed rectangles (dashed boxes) represents the weighted percentage of faults detected over the corresponding fraction of the test suite. The solid lines connecting the corners of the inscribed rectangles interpolate the gain in the percentage of detected faults. This interpolation is a granularity adjustment when only a small number of unit tests comprise a test suite; the larger the test suite the smaller this adjustment. The area under the curve thus represents the weighted average of the percentage of faults detected over the life of the test suite. This area is the prioritised test suite's average percentage faults detected measure (APFD); the APFD is 50 in this example. Figure 8.2 shows the effects of using a prioritised test suite PTS_2 whose test case ordering is [ut_3-ut_5-ut_2-ut_1-ut_4].

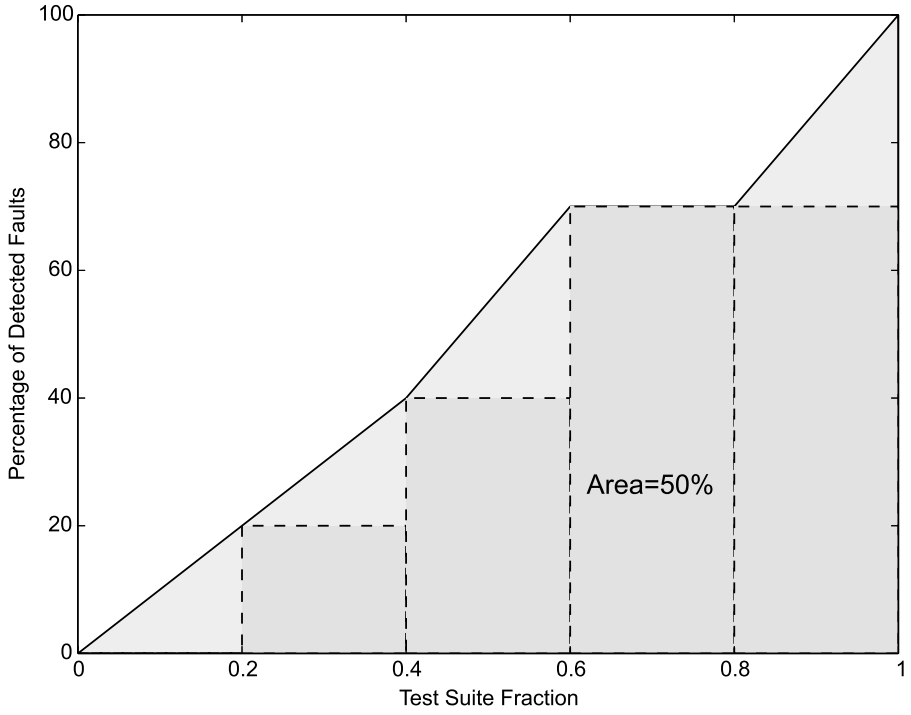


Figure 8.1: APFD for prioritised test suite PTS_1

By inspection, it is clear that this ordering results in the earliest detection of the most faults and illustrates an optimal ordering, with APFD 84.

It is worth noting that this metric assumes that all faults have the same severity and that all unit test executions have the same cost. Another more sophisticated measure is presented in [EMR01] which allows to assign cost and severity to unit tests and faults respectively. Still, we use the standard APFD metric because (*i*) in each case study the execution cost of the unit tests is approximately the same; and (*ii*) because we introduce faults by systematically applying mutation operators and we cannot determine the severity of each different mutated program without introducing bias.

8.3 Coverage-Based Prioritisation Techniques

Prioritisation-based on coverage is based on the hypothesis that early maximisation of coverage can lead to early detection of faults. Since location of faults is a priori unknown, prioritisation techniques schedule unit tests for increasing their effectivness according to some metric that acts as a proxy of fault detection. For instance, unit tests can be scheduled for achieving code, action or requirement coverage at the fastest rate possible. The assumption here is that the test suite is ordered based on some criterion that is expected to lead to the early detection of faults.

There are many coverage-based prioritisation techniques that have been proposed and evaluated in the literature [YH12]. They can be divided in two groups: on the one hand those that consider total coverage and on the other hand those that consider additional coverage. More specifically:

- TOTAL COVERAGE-BASED TECHNIQUES: Techniques prioritise unit test based

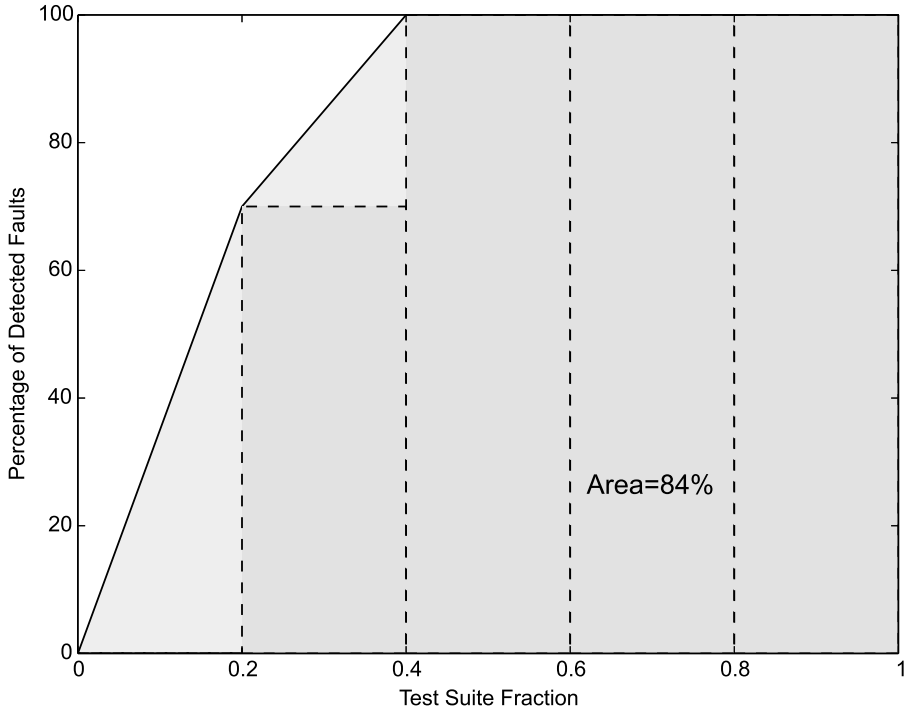


Figure 8.2: APFD for prioritised test suite PTS_2

on their total individual coverage. That is, unit tests that cover more parts of the system, based on a chosen coverage criterion, are ordered before those that cover less. If several unit tests have the same coverage, the order is determined randomly.

- Additional coverage-based techniques: The additional coverage techniques order unit tests so that coverage is maximised as early as possible. The technique differs from the total coverage approach in that the next unit test is selected based on the highest coverage of uncovered parts rather than the highest coverage of the whole. If there is more than one test that add the same coverage of yet uncovered parts, one is randomly picked.

Recent experimental work on coverage-based prioritisation techniques has provided some evidence that additional coverage-based techniques produce better orderings than those obtained by using total coverage-based ones [NABL13]. Therefore, we focus here on additional coverage-based techniques. Concretely they are:

- Additional EPA transition coverage: Unit tests are ordered in a way such that the next unit test is the one that covers more yet uncoverd transitions.

- Additional EPA action-transition coverage: Unit tests are ordered in a way such that the next unit test is the one that covers more yet uncoverd actions and transitions, giving priority to actions over transitions (in the same way as in definition 8).

- Additional statement coverage: Unit tests are ordered in a way such that the next unit test is the one that covers more yet uncoverd statements of the code.

- ADDITIONAL BRANCH COVERAGE: Unit tests are ordered in a way such that the next unit test is the one that covers more yet uncoverd branches of the code.

Additionally, we also calculate the APFD values of randomly-ordered test suites.

8.4 Data Collection

For each of the five case studies we use the information recorded during the experimentation described in Chapter 3, in which unit tests were executed automatically and the outputs were saved for later analysis.

The number of unit tests that compose a test suite combined with the difficulty of finding the faulty versions may have an important impact on APFD values. Let us consider for instance the prioritised test suite PTS_1 of Figure 8.1, and suppose that after t_5 we append 95 additional unit tests. Since after executing t_5 all faulty versions have already been exposed, independently of how good these new unit tests are, the APFD value increases a lot (the APFD of this new prioritised test suite is 97.5 while the original is 50). In general, adding unit tests after all faulty versions have been found makes APFD values grow. As described in chapter 3, for each case study we generated 10000 unit test. Considering that an important fraction of the mutants generated are not really hard to discover, orderings of 10000 unit tests would always yield very high APFD values. Therefore, instead of considering orderings of the 10000 unit tests, for each case study we generate 30 different test suites following the procedure described by Algorithm 1. The algorithm starts by creating an empty test suite, and then randomly adds unit tests to it until the test suite is capable of detecting all faulty versions. This allows us to generate, for each case study, different test suites of appropriate sizes for analizing APFD values. The average size (as the number of unit tests) of the resulting test suites for `ListIterator`, `Signature`, `Socket`, `JDBCResultSet` and `SMTPProcessor` are 22, 29, 8, 61 and 58 respectively.

ALGORITHM 1: Generation of test suites that discover all faulty versions

Input: All unit tests
Output: A test suite that reveals all faulty versions
$TestSuite = []$
while *not all faulty versions have been detected* **do**
  randomly select one unit test *ut*
  if $ut \notin TestSuite$ **then**
    append($TestSuite$,*ut*)
  end
end
return *TestSuite*

As above mentioned, there is a random component in the studied techniques. While making an ordering, it may be the case that the highest additional coverage is achieved by more that one unit test, so one is randomly selected. In order to mitigate the accidental influence of such random choices, for each of the 30 test suites we generate 30 different orderings according to each of the four additional coverage-based techniques considered.

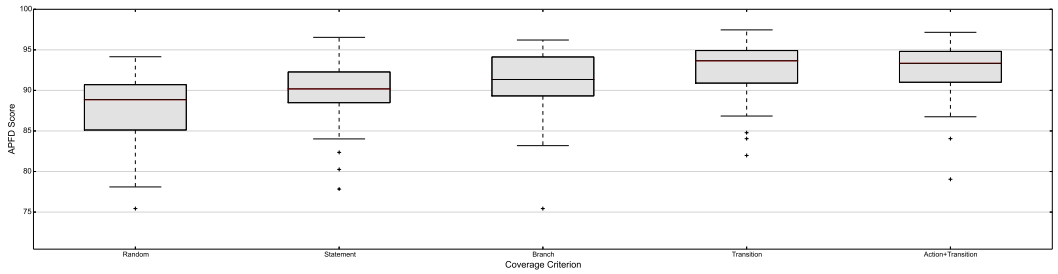
8.5 Results

Figure 8.3 shows boxplots of the APFD values obtained in each of the five case studies. The boxplots show the distribution of APFD values of random orders and the additional coverage-based techniques considered: Additional statement coverage, additional branch coverage, additional EPA transition coverage and additional EPA action-transition coverage. Numerical values of percentils, means and means excluding outliers are shown in table 8.2.

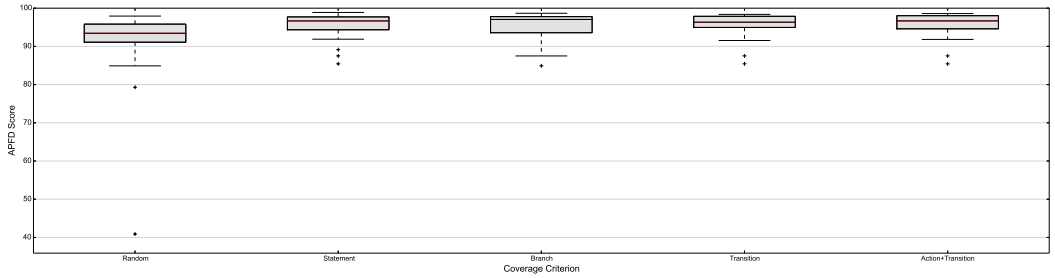
Subject	Prioritisation technique	Q1	Median	Q3	Mean	Mean EO
`ListIterator`	Random	85.1	88.92	90.85	87.06	87.46
	Statement	88.46	90.27	92.31	89.65	90.4
	Branch	89.18	91.44	94.19	90.9	91.44
	EPA Transtion	90.89	93.91	94.94	92.4	93.38
	EPA Action-Transition	90.95	93.39	94.83	92.35	93.12
`Signature`	Random	90.92	93.49	95.83	91.11	92.84
	Statement	94.34	96.84	97.74	95.29	95.92
	Branch	93.23	97.11	97.81	95.51	96.18
	EPA Transtion	94.95	96.46	97.88	95.42	96.07
	EPA Action-Transition	94.51	96.88	98.06	95.68	96.34
`Socket`	Random	74.86	79.94	84.58	78.93	79.48
	Statement	80.3	83.69	87.63	82.87	83.98
	Branch	75.64	82.2	87.29	81.09	81.6
	EPA Transtion	75.99	83.19	87.75	81.75	82.27
	EPA Action-Transition	76.55	82.2	86.86	81.53	82.05
`JDBCResultSet`	Random	95.23	96.12	96.91	96.0	96.11
	Statement	96.51	97.31	98.67	97.46	97.46
	Branch	96.88	97.59	98.28	97.52	97.52
	EPA Transtion	97.38	98.2	98.55	97.98	97.98
	EPA Action-Transition	97.06	97.72	98.57	97.77	97.84
`SMTPProcessor`	Random	80.97	81.32	85.88	84.28	83.37
	Statement	85.39	92.62	96.52	91.37	91.37
	Branch	85.38	90.76	95.14	90.15	90.15
	EPA Transtion	86.56	93.48	96.58	91.91	91.91
	EPA Action-Transition	87.77	89.74	93.88	90.69	90.69

Table 8.2: APFD values percentiles and average

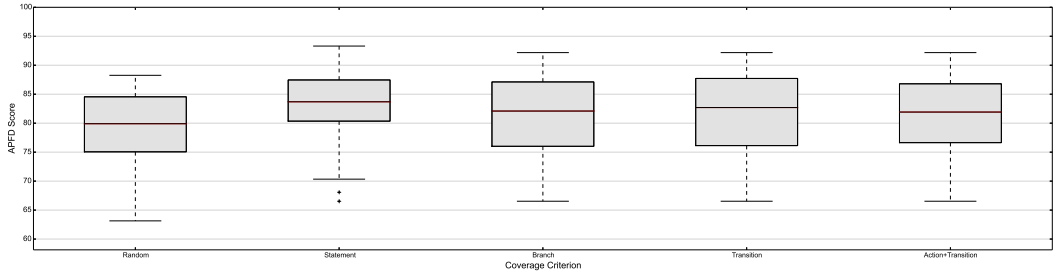
As expected, APFD values based on additional coverage techniques are greater than those of random orderings. In the case of `ListIterator`, best values are in all cases those corresponding to orderings based on additional EPA Transition coverage and additional EPA Action-Transition coverage. Moreover, the Q1 value of the orderings based on EPA Action-Transition coverage is greater than mean values of those based on covering statements and branches (i.e., at least 75% of the orderings based on EPA Action-Transition coverage have a faster fault detection capability than average orderings based on covering code). Almost the same observation is true for values corresponding to EPA Transition, except for the case of the mean value of branch coverage, which is slightly (0.01) greater. As already mentioned in previous chapters, `ListIterator` has a rich protocol and a very simple code. Therefore, ordering unit tests based on EPA coverage -which means covering transitions as soon as possible- leads to a faster detection of faulty version than those based on code coverage, which is consistent with all the results presented along this thesis. Also in the case of `SMTPProcessor` best APFD values are those obtained by orderings based on EPA Transition and EPA Action-Transition additional coverage. In the case of `Signature`, greatest APFD values of Q1, Q3, mean and mean excluding outliers correspond again to orderings based on EPA coverage, whereas the highest median is for orderings based on branch coverage. For `JDBCResultSet`, greatest APFD values of Q1, median, mean and mean excluding outliers are also those of the orderings based on EPA Coverage, while highest Q3 is that of statement coverage. The case of `Socket` shows a different trend: except for Q3 where orderings based



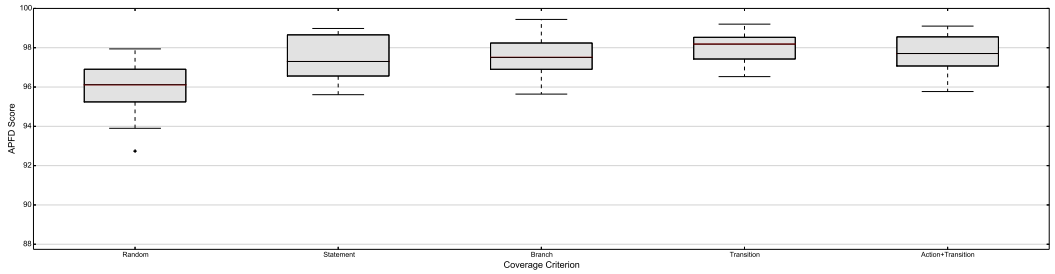
(a) `ListIterator`



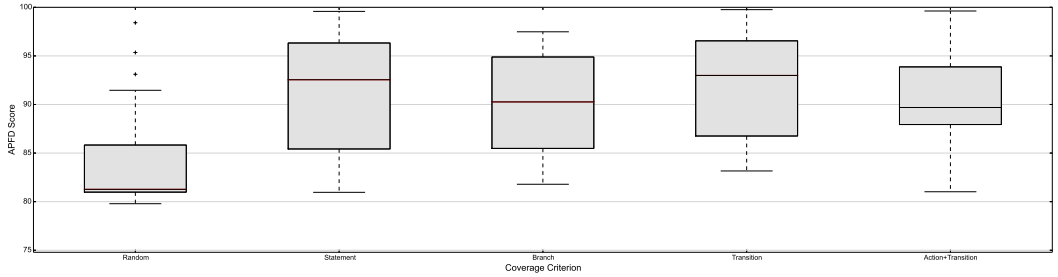
(b) `Signature`



(c) `Socket`



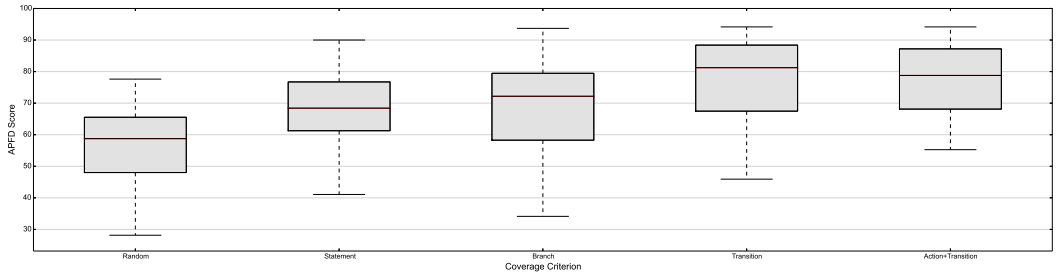
(d) `JDBCResultSet`



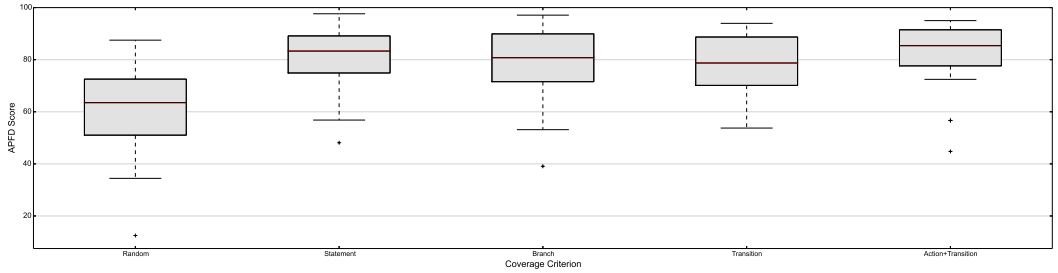
(e) `SMTPProcessor`

Figure 8.3: Distribution of the APFD values

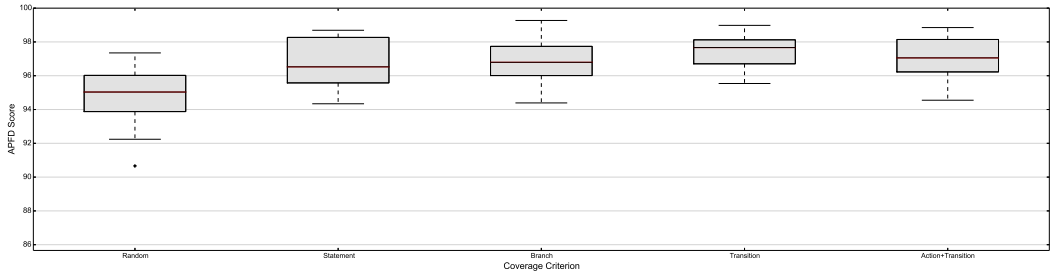
on EPA Transition achive the highest value, in all others cases the greatest values are those of transition coverage. Summarizing, in general APFD values for criteria based on EPA coverage are greater than those corresponding to code coverage.



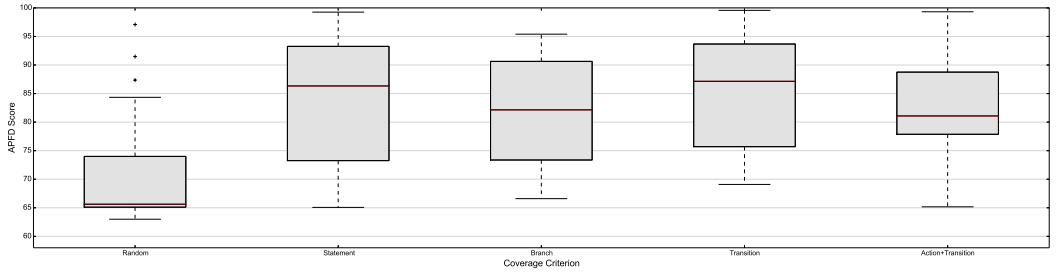
(a) `ListIterator`



(b) `Signature`



(d) `JDBCResultSet`



(e) `SMTPProcessor`

Figure 8.4: Distribution of the APFD values considering only hard mutants

We also calculate the APFD values restricting the set of faulty implementations to those that are hard to detect (i.e., that are detected by less that 20% of the unit tests as shown in table 5.5). Boxplots of the results are shown in figure 8.4 and numerical values in table 8.3.

Interestingly, when restricting the universe of mutants, all highest values except one correspond to orderings based on additional EPA transition coverage or additional EPA action-transition coverage (the only exception is the value of Q3 of case study `JDBCResultSet`, where highest value is that of additional statement coverage).

In the case of `ListIterator`, highest value of Q1 is the one of EPA action-transition coverage, and highest values for median, Q3, mean and mean excluding outliers are those of EPA transition coverage. It is worth noting that in general the differences of APFD values between orderings based on covering EPAs and

Subject	Prioritisation technique	Q1	Median	Q3	Mean	Mean EO
`ListIterator`	Random	47.35	58.93	65.58	56.99	57.99
	Statement	61.11	68.65	76.79	66.73	66.73
	Branch	58.16	72.29	79.76	68.78	69.98
	EPA Transtion	66.48	83.33	88.69	77.09	79.24
	EPA Action-Transition	67.86	80.1	87.5	76.91	76.91
`Signature`	Random	50.79	63.85	72.65	59.71	61.34
	Statement	74.62	83.57	89.67	80.82	82.84
	Branch	71.33	82.82	90.0	79.29	80.67
	EPA Transtion	69.49	78.89	89.11	78.6	80.37
	EPA Action-Transition	77.31	86.2	91.54	82.21	85.49
`JDBCResultSet`	Random	93.86	95.05	96.03	94.87	95.02
	Statement	95.51	96.53	98.29	96.73	96.73
	Branch	95.98	96.89	97.79	96.81	96.81
	EPA Transtion	96.65	97.69	98.15	97.41	97.41
	EPA Action-Transition	96.21	97.07	98.17	97.14	97.23
`SMTPProcessor`	Random	65.09	65.72	74.09	71.15	69.5
	Statement	73.2	86.47	93.62	84.19	84.19
	Branch	73.18	83.05	91.1	81.94	81.94
	EPA Transtion	75.34	88.05	93.73	85.16	85.16
	EPA Action-Transition	77.56	81.17	88.77	82.91	82.91

Table 8.3: APFD values percentiles and average

orderings based on structural coverage are greater now than those obtained when considering the universe of all mutants. In the case of `Signature` all highest values are those of additional EPA action-transition coverage. For `JDBCResultSet` highest values for Q1, median, mean and mean excluding outliers are those of additional EPA transition coverage, and as above mentioned the highest Q3 value is that of additional transition coverage. Last, in the case of `SMTPProcessor` Q1 highest value correspond to additional EPA action-transition coverage, and in all the remaining cases to EPA transition coverage.

Summarizing, ordering unit test of a test suite that maximise as soon as possible the coverage of the behaviour abstraction tend to produce faster rate of fault detection than those that prioritise structural coverage.

Part IV

Discussion

CHAPTER 9

Threats to Validity

The results presented in this thesis are subject to threats to validity. In this chapter we discuss them. We distinguish between threats to internal, external and construct validity.

9.1 Threats to External Validity

Threats to external validity concern our ability to generalise the results. The proposed approach is not suitable for testing code which has trivial requirements regarding the order in which methods or procedures must be called. Furthermore, even fixing the code to that which has rich intended protocols, the results cannot be generalised to identification of faults other than the ones captured by the failure model embedded in the notion of call protocol conformance discussed in this work. The expectation is that the results can be generalised to classes featuring rich intended protocols and faults that are expressed as unexpected occurrence of exceptions or non-terminating methods. However, the study presented only covers five subjects which may not be sufficiently representative of rich protocol code artefacts. Moreover, restricting faulty implementations to those that unexpectedly raise exceptions or timeout is limitating, since APIs may fail in other ways too. Still, despite its weakness it is still considered relevant by the software engineering community in typestate verification (e.g. [BKA11, FGRY05, FYD+08, BA07]) and concurrent systems (e.g. [Sch99, MK06]).

A threat to the generalisation our results to the breadth of EPA coverages is that for all subject there are transitions that were not covered by any test suite (coverages achieved in subjects are 82%, 93%, 90%, 81%, and 93% for `SMTPProcessor`, `ListIterator`, `Socket`, `JDBCResultSet`, and `Signature` respectively). Not achieving total coverage is due, firstly, to the use of random generation of test units which makes reaching "deep" EPA states unlikely. Possibly a much larger number of much longer unit tests could have achieved more coverage. Secondly, there is an essential difficulty in reaching deep states due to the complexity of code under test. In fact, these problems appear when trying to achieve high structural coverage in general [XXTdH11], and for the subjects chosen in particular. Guided test-case generation could have been used to address this issue, however the threat of a biased pool of test suites would have increased.

Another threat to generalisation is due to our choice of restricting the scope of

this work to deterministic APIs. Extending our work to implementations that may exhibit non-deterministic failures would require to revisit the coverage criteria and the conformance relation. Besides, it would certainly require a more sophisticated experimental strategy. If non-deterministic API implementations were included, the notion of test effectiveness (killing a mutant) would require probabilistic treatment since a non-deterministic mutant could behave differently in terms of failing or not to accept a given sequence of calls. Therefore, each test would have a probability of killing each mutant.

9.2 Threats to Internal Validity

Threats to internal validity appear as a consequence of how we conducted the experiments. One major threat is the validity of the EPAs of the actual protocol LTS for the subjects studied. The use of LTS that do not abstract appropriately the behaviour of the subject implementations could lead to skewed results regarding coverage (although not for detecting faults, as for this the reference implementation itself and not its abstraction is used). We believe that the risk of having used models that are not proper abstractions of the subjects (i.e. not EPAs) is mitigated by our systematic construction process, validation against third party constructed models and manual inspections performed.

Another threat to internal validity appears as a consequence of the adopted fault model. We introduce faults using a mutation tool. Whether mutation analysis is a realistic fault model or not is still open and beyond the scope of this paper. Nevertheless, some work has shown statistical evidence of the validity of using mutation analysis for the evaluation of testing techniques [ABL05], and it is a very widespread approach for evaluating testing techniques.

Another threat to internal validity is due to the adopted failure model. Incorrect computations can go unnoticed and exceptions can be triggered later: this is a problem that affects unit test in general and not something particular of this work. The goal is, in this case, to see if the sequences generated according to our criteria produce inputs that not only reach the state infection but also its manifestation. It could be the case, that reduced oracle power might worsen this general phenomena and thus imply the need of longer test cases to detect manifestation but this is mitigated by using a set of longer unit tests.

As with other experiments using mutants and unit tests, the threat of using weak tests that fail to identify fault-inducing mutants exists. This could lead to different results if these harder to kill mutants were included. However we have analysed correlations over subsets of mutants found. In particular, those least killed showed no significant changes in correlation.

We believe that threats regarding unintended effects of general experimental infrastructure needed for mutant generation and fault detection are minor since we have used standard tools such as μ-Java and Randoop whenever possible and simple code instrumentation techniques using AspectJ to record mutant detection.

9.3 Threats to Construct Validity

Threats to construct validity mainly appear from our choice to compare our criterion with code coverage criteria. We recognise that code coverage as a measure of effectiveness of a test suite is still being studied by the testing community. However, in order to asses if the correlations with fault detection for EPA coverage are

reasonable, some well accepted baseline is needed. We chose structural code coverage criteria. However, we also complement comparisons with code coverage in our experiments with the study of EPA coverage against fault detection.

On the other hand, the proposed criteria are black box while we compare with what in principle could be considered white box criteria in RQ1 (i.e. structural code coverage). It could be argued that a more suitable baseline would be some other black box criteria such as structural coverage over a specification. The question here would be what specification language would be suitable. Given that the intended protocol LTS of the subjects that are of interest for this study are infinite state, a rich specification language such as Extended Finite State Machines (essentially, LTS with variables) as used in [OLAA03, KKT07] is required, or even combined with programming languages such as C# or Java -as used in model based testing approaches such as SpecExplorer [VCG+08] and ConformiQ [UL07]. Unfortunately, there is no de facto standard black box baseline for rich modeling languages and any choice of language, tool and specification style will introduce bias as it is known that specification structure can have significant impact on coverage criteria adequacy (e.g., [HGW04, RHOH98, PPW+05, RCW08, MBLDP11, Wei10], etc.). We believe that taking the most detailed specification (the code itself) constitutes an upper-bound on what structural criteria on specifications can achieve as test-suite quality predictor. In fact, it is highly likely that in Model Based Development there will be more structural discrepancy between specification and code under test. Hence, it could be argued that choosing the code as the specification hinders validation of the hypothesis being proposed in this thesis. The use of a reference implementation as the specification of the intended protocol LTS that is to be provided by the mutated implementations, we believe it reduces the structural discrepancy between specification and implementation and do not bias the result in favor of the hypothesis studied in this thesis. That is why RQ1 and RQ2 can be also thought of as comparing the results against black-box too under the previous caveats.

CHAPTER 10

Related work

In this Chapter we compare the work presented herein with others. The chapter is divided into two sections. In Section 10.1 we acknowledge relevant work which also use models in the context of software testing; and in Section 10.2 we focus on diverse approaches that specifically tackle the problem of protocol conformance.

10.1 Models for Testing

Generation of test sequences for software with internal state has captured the attention of researchers in recent years [Ton04, AML11, IX08, VPP06, LMS07]. Those works aim at generating internal states that improve structural coverage by constructing a set of test sequences. Pure Random Testing approaches like [PLEB07] are based on reusing previously returned values. Those approaches are truly black box and thus are applicable to conformance testing and off-line test generation. In fact, [PLEB07] constitutes our baseline for the experiments conducted. Evolutionary approaches like [Ton04] represent initial randomly generated method sequences as a population of individuals and evolve this population by mutating its individuals until a desirable set of method sequences is found. Although they do not use program structure or semantic knowledge to directly guide test generation, they do require coverage achieved by test sequences on SUT for computing the fitness function. This makes them less portable to our problem setting. Recent genetic techniques instead aim at finding settings to parameters which control aspects of randomized testing (e.g., [AML11]). The systemic and randomized method of [IX08] is even less applicable to the problem setting because it requires access to program code of SUT to generate inputs by demand by the use of some symbolic computation approaches.

Much research effort on testing has focused mainly on test case generation by exploiting to various degrees the code-under-test: from purely systematic white-box approaches (e.g., [TXT+11]) to search-based approaches [SGA+11] in which fitness functions are based on achieved coverage. None of these approaches can tackle conformance checking to its full extent: they are not driven by any form of actual or intended behaviour. However, some works explicitly or implicitly define or mine models to improve the quality of tests. For instance, in [LMS07] the state space of a class is quotiented based on its parameterless boolean observers (similar approach for a different purpose is in [XN04]). In [VPP06], abstract states are computed using shape abstraction, i.e., ignoring the concrete values in containers and taking

into account only the shape in which the container nodes are connected. Note that this work requires access to internal state of the SUT. In [DKM+10], a type-state model -similar to our EPA- is inferred and used to guide the generation of new test cases that try to cover uncovered transition of the type state. The goal is to dynamically discover typestate models. The quality of such models is then measured in the context of detecting misuse of the class protocol by client programs.

The work presented in this thesis differs substantially in various ways: The mentioned approaches (*i*) do not look at the problem of black-box conformance testing; (*ii*) do not articulate equivalence criteria declaratively as an adequacy criterion (rather, they are tightly coupled to the particular technique); and (*iii*) do not provide statistical evidence on the effectiveness of covering such abstractions in terms of fault rate detection.

10.2 Conformance Testing

There has been plenty of work focused on defining coverage from formal specifications and models [HBB+09]. Works range on a plethora of languages: algebraic specifications [GMH81, DF94, BGM91], Z [Hie97], VDM [DF93], boolean specifications [WGS94], tabular notations [GH99], reactive modeling language like SDL [BPBM97], EFSM (or satchecharts) [PPW+05, OLAA03, PBG04], sequence charts [DH11] temporal logics [PRB09, AB99, HLSU02], C# [VCG+08], ADL [MIB04], simulation code [RCW08], etc. For the rest of the section, we focus the discussion on approaches able to straighforwardly aim at testing object protocols.

Existing approaches can be classified in two categories: structural (or specification-based) and behaviour (or semantic) coverage criteria. Structural coverage criteria are either defined in terms of the specification (e.g., [BPBM97, KKT07, OLAA03, Hie97, DF93, PRB09], etc.) or of the executable code generated from the specification or simulation model (e.g.[PPW+05, RCW08]). Representative examples are [OLAA03, KKT07, MBLDP11] where criteria are defined over syntactic elements as transitions and predicates featured in expressive state-based specification languages like EFSMs or UML state machines.

Although empirical studies looking for statistical evidence on the suitability of coverage criteria are rather common for code coverage criteria (e.g., [FW93a]) yet are scarce for state-based specification languages [HBB+09] which are particularly appropriate for conformance testing. Some notable exceptions we found are [RCW08, PPW+05, MBLDP11]. Interestingly enough, in these, experiments were conducted on criteria based on covering code generated from models [PPW+05] or simulation code [RCW08]. In [MBLDP11] hand-made test suites based on UML state machines are compared against test suites based on structural testing. Like authors of [PPW+05], we speculate that difficulties on automation criteria over specification could be a symptom of a lack of comprehensive definitions and tools for specification-based coverage criteria for rich state-based languages. We believe the work presented herein is a step forward in this direction.

On the other hand, in behavioural approaches, coverage is defined in terms formalisms which straightforwardly denote the intended protocol behaviour. This line of work is that of seminal work on black box testing in the context of Finite State Machine and protocol testing [LY96]. In foundational work, the conformance problem is stated in terms of Mealy machines. However, in contrast to our approach, coverage and failure models assume finiteness of both the specification and the actual implementation.

A work that drops the finiteness assumption is that of LTS based testing [JT96]

where IOCO is a well established notion of conformance. In IOCO it is required that for any valid sequence (according to the specification), the implementation's outputs after the sequence should be a subset of the specified outputs. In our work, if we consider valid outputs as {`OK, ERROR`}, then our notion of conformance is the same as the one used in IOCO: If the specification says `OK`, then the implementation must say `OK` too; and if the specification says `ERROR` also the implementation should do the same. However, no notion of behaviour coverage has been defined in this setting of infinite state space.

A way of dealing with infinite behaviour models is by introducing abstraction. This is the case for some application domains of testing, like protocol testing [BP94] and architecture-based testing [MIB04], where the level of abstraction in which the designer expresses the specification leads to the finiteness of underlying LTS semantics. However, if the underlying semantic model is truly infinite, then some sort of finitisation is made available to the tester, etc. Several finitisation techniques exist: unfolding [BPBM97], domain bounding and slicing [VCG+08, GKSB11], and state prunning [GKSB11]. However, no statistical studies on coverage for these finitisations are available. Other relevant work in this line is that of automatic under approximation of infinite behaviour from concrete [MPRS11] or symbolic [GGSV02] executions that can be later used for regression testing. Here, again, no statistical study is available.

In [MV95] authors use the term "cover" in the topological sense defining a finite number of subspaces that cover an infinite metric space. Covering (in the testing sense) a representative of each of these subspaces amounts to defining a coverage criterion over an abstraction of the infinite state space, which is what we also do. As discussed in this thesis, the positive impact of partitioning an input space to defining coverage criteria is highly dependent on how failures fall within these partitions. Thus, experimental evaluation of coverage criteria is important. We have not found experimental evaluation of the predictive power of criteria defined over finite abstractions of infinite input spaces for API call protocols.

Summarising, the approach statistically studied in this thesis fits in the category of behaviour or semantic approaches to conformance testing in which infinite state space is handled by means of an abstraction that over-approximates the state space much in the vein of typestates [BBA09]. This abstraction produces a partition of states and transitions thus constituting a case of category partition [OB88] for infinite LTS.

CHAPTER 11

Conclusions and Future Work

This thesis is a step towards defining and understanding how semantic coverage of infinite state behaviour specifications relates to effective testing techniques for call protocol conformance. We address this by studying coverage achieved on an abstraction of such behaviour, more specifically on enabledness preserving abstractions (much in the vein of typestates [BBA09]). We believe a good understanding of the relation between fault detection, white box coverage criteria, and coverage of abstractions of the semantic space of specifications could help to improve random testing, test case selection, prioritization techniques and, in general, heuristics to generate tests from formal specifications.

The results we obtained in the experiments reported in this thesis are promising and suggest that EPA coverage performs well in term of predictability of test-suite fault detection. This is particularly important in a black box testing setting and it constitutes an opportunity for defining criteria that are independent of modeling notation and accidental characteristics of models themselves. Results also suggest that EPA coverage criterion can make a difference in terms of fault detection for tests suites of the same length.

In addition, results lead to believe that EPA coverage is a good predictor of statement and branch coverage, and that, for same sized test suites high EPA coverage is more likely to achieve high code coverage. This may have practical implications in the context of development approaches which advocate test development before coding (e.g. test driven development, interoperability, etc.) or automated generation of test suites (model driven development). In these contexts, developing tests according to the EPA of the intended protocol would allow a first (and early) shot at producing high code coverage test suites. These test suites could later be extended, if necessary, when code is available. As far as we know, existing approaches for generating high code coverage test suites are all white box and therefore, they typically need the source code to be applied. It is important to note that the construction of EPA abstractions of intended protocol behaviour is feasible, practical and tool supported from contract-based specifications [dCBGU09] or code [dCBGU11]. Furthermore, EPA abstractions could be provided directly by testers as advocated by typestate approaches [BKA11].

Besides, results show that the domain partition implicitly derived from EPA transitions tends to produce subdomains that are dense in failures, which is near to an ideal scenario from a category partition viewpoint [JW89]. Furthermore,

we observe that for each fault there is almost always one transition that is highly effective in detecting it, and that the effective transition varies from fault to fault. Additionally, we show that the possitive results are not just a consequence of covering actions, and that when both actions and transitions are taken into account the resulting criterion has highest correlation values regarding fault detection.

The results could also be the basis for arguing for test case generation strategies based on abstractions of behaviour rather than code itself (which requires significantly more heavyweight machinery). In addition, we believe that they also make a case for stating that test case generation for rich call protocol artefacts (from random black box to sophisticated concolic white box techniques) can benefit from the heuristics that are implied by the results discussed in this thesis.

Future work should aim at looking at other protocol abstractions and comparing them with EPAs in terms of their effectiveness for testing call protocol conformance. Besides, we want to conduct experiments in a more general setting that includes both APIs with non-deterministic expected behaviour and output values as part of the conformance relation considered. We also plan to study cost/benefit analysis when these ideas are instantiated to a guide the generation of test suites. In fact, we speculate, random generation could benefit from EPAs not only due to the results shown in this thesis but also the availability of an abstract protocol would help in implementing heuristics aimed at the early execution of particular actions or functionalities.

Bibliography

[AB99] Paul E. Ammann and Paul E. Black. A specification-based coverage metric to evaluate test sets. *HASE 99*, 0:239, 1999.

[AB11] Andrea Arcuri and Lionel C. Briand. A practical guide for using statistical tests to assess randomized algorithms in software engineering. In *ICSE '11*, pages 1–10, 2011.

[ABL05] J. H. Andrews, L. C. Briand, and Y. Labiche. Is mutation an appropriate tool for testing experiments? In *ICSE '05*, pages 402–411, 2005.

[ABLN06] James H. Andrews, Lionel C. Briand, Yvan Labiche, and Akbar Siami Namin. Using mutation analysis for assessing and comparing testing coverage criteria. *IEEE Transactions on Software Engineering*, 32(8):608–624, 2006.

[AGWX08] J.H. Andrews, A. Groce, M. Weston, and Ru-Gang Xu. Random test run length and effectiveness. In *ASE '08*, pages 19–28, 2008.

[AML11] James H. Andrews, Tim Menzies, and Felix C.H. Li. Genetic algorithms for randomized unit testing. *IEEE Transactions on Software Engineering*, 37(1):80–94, 2011.

[AZ03] James H Andrews and Yingjun Zhang. General test result checking with log file analysis. *IEEE Trans. Softw. Eng.*, 29(7):634–648, 2003.

[BA07] Kevin Bierhoff and Jonathan Aldrich. Modular typestate checking of aliased objects. *SIGPLAN Not.*, 42(10):301–320, 2007.

[BBA09] Kevin Bierhoff, Nels E. Beckman, and Jonathan Aldrich. Practical api protocol checking with access permissions. In *Proceedings of the 23rd European Conference on ECOOP 2009 — Object-Oriented Programming*, Genoa, pages 195–219, 2009.

[Bec03] K. Beck. *Test-driven development: by example*. Addison-Wesley Professional, 2003.

[BGM91] Gilles Bernot, Marie Claude Gaudel, and Bruno Marre. Software testing based on formal specifications: a theory and a tool. *Software Engineering Journal*, 6(6):387–405, 1991.

[BKA11] N.E. Beckman, D. Kim, and J. Aldrich. An empirical study of object protocols in the wild. In *ECOOP '11*, pages 2–26, 2011.

[BLW04] L. C. Briand, Y. Labiche, and Y. Wang. Using simulation to empirically investigate test coverage criteria based on statechart. In *ICSE '04*, pages 86–95, 2004.

[BP94] Gregor V. Bochmann and Alexandre Petrenko. Protocol testing: review of methods and relevance for software testing. In *ISSTA '94*, pages 109–124, 1994.

[BPBM97] G. Bochmann, A. Petrenko, O. Bellal, and S. Maguiraga. Automating the process of test derivation from SDL specifications. In *SDL Forum '97*, 1997.

[CBGU13] Guido De Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Enabledness-based program abstractions for behavior validation. *ACM Transactions on Software Engineering and Methodoly*, 22(3):25:1–25:46, 2013.

[CYH06] M. Cihan Yalcin and Yenigun H. Using distinguishing and uio sequences together in a checking sequence. In *TESTCOM '06*, pages 259–273, 2006.

[DASC93] Khalil Drira, Pierre Azéma, B. Soulas, and A. M. Chemali. Characterizing and ordering errors detected by conformance testing. In *IWPTS '92*, pages 67–78, 1993.

[dCBGU09] Guido de Caso, Victor Braberman, Diego Garbervetsky, and Sebastian Uchitel. Validation of contracts using enabledness preserving finite state abstractions. In *Proceedings of the 31st International Conference on Software Engineering*, ICSE '09, pages 452–462, 2009.

[dCBGU11] Guido de Caso, Víctor Braberman, Diego Garbervetsky, and Sebastián Uchitel. Program abstractions for behaviour validation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 381–390, 2011.

[DF93] Jeremy Dick and Alain Faivre. Automating the generation and sequencing of test cases from model-based specifications. In *FME '93: Industrial-Strength Formal Methods*, LNCS, pages 268–284. 1993.

[DF94] Roong-Ko Doong and Phyllis G. Frankl. The astoot approach to testing object-oriented programs. *ACM Trans. Softw. Eng. Methodol.*, 3:101–130, 1994.

[DF04] Robert DeLine and Manuel Fähndrich. Typestates for objects. In *ECOOP '09*, pages 465–490, 2004.

[DH11] Haitao Dan and Robert M. Hierons. Conformance testing from message sequence charts. *Software Testing, Verification, and Validation, 2008 International Conference on*, 0:279–288, 2011.

[DKM+10] Valentin Dallmeier, Nikolai Knopp, Christoph Mallon, Sebastian Hack, and Andreas Zeller. Generating test cases for specification mining. In *ISSTA '10*, pages 85–96, 2010.

[EMR00] Sebastian Elbaum, Alexey G. Malishevsky, and Gregg Rothermel. Prioritizing test cases for regression testing. *SIGSOFT Softw. Eng. Notes*, 25(5):102–112, August 2000.

[EMR01] Sebastian Elbaum, Alexey Malishevsky, and Gregg Rothermel. Incorporating varying test costs and fault severities into test case prioritization. In *Proceedings of the 23rd International Conference on Software Engineering*, ICSE '01, pages 329–338, Washington, DC, USA, 2001. IEEE Computer Society.

[FA12] G. Frase and A. Arcuri. A sound empirical evidence in software testing. In *ICSE '12*, pages 178–188, 2012.

[FGRY05] J. Field, D. Goyal, G. Ramalingam, and E. Yahav. Typestate verification: Abstraction techniques and complexity results. *Science of Computer Programming*, 58(1-2):57–82, 2005.

[FW93a] P. G. Frankl and S. N. Weiss. An experimental comparison of the effectiveness of branch testing and data flow testing. *TSE*, 19:774–787, 1993.

[FW93b] Phyllis G. Frankl and Elaine J. Weyuker. A formal analysis of the fault-detecting ability of testing methods. *TSE*, 19:202–213, 1993.

[FYD+08] Stephen J. Fink, Eran Yahav, Nurit Dor, G. Ramalingam, and Emmanuel Geay. Effective typestate verification in the presence of aliasing. *TOSEM*, 17(2):9:1–9:34, 2008.

[GGSV02] Wolfgang Grieskamp, Yuri Gurevich, Wolfram Schulte, and Margus Veanes. Generating finite state machines from abstract state machines. In *ISSTA '02*, pages 112–122, 2002.

[GGZ+13] Milos Gligoric, Alex Groce, Chaoqiang Zhang, Rohan Sharma, Mohammad Amin Alipour, and Darko Marinov. Comparing non-adequate test suites using coverage criteria. In *ISSTA '13*, pages 302–313, 2013.

[GH99] Angelo Gargantini and Constance Heitmeyer. Using model checking to generate tests from requirements specifications. In Oscar Nierstrasz and Michel Lemoine, editors, *Software Engineering - ESEC/FSE '99*, volume 1687 of *Lecture Notes in Computer Science*, pages 146–162. Springer Berlin / Heidelberg, 1999.

[GKSB11] Wolfgang Grieskamp, Nicolas Kicillof, Keith Stobie, and Victor Braberman. Model-based quality assurance of protocol documentation: tools and methodology. *STVR*, 21(1):55–71, 2011.

[GMH81] John Gannon, Paul McMullin, and Richard Hamlet. Data abstraction, implementation, specification, and testing. *ACM Trans. Program. Lang. Syst.*, 3:211–223, 1981.

[GZE+12] Alex Groce, Chaoqiang Zhang, Eric Eide, Yang Chen, and John Regehr. Swarm testing. In *ISSTA '12*, pages 78–88, 2012.

[HBB+09] Robert M. Hierons, Kirill Bogdanov, Jonathan P. Bowen, Rance Cleaveland, John Derrick, Jeremy Dick, Marian Gheorghe, Mark Harman, Kalpesh Kapoor, Paul Krause, Gerald Lüttgen, Anthony J. H.

Simons, Sergiy Vilkomir, Martin R. Woodward, and Hussein Zedan. Using formal specifications to support testing. *ACM Computing Surveys*, 41:9:1–9:76, 2009.

[HGW04] Mats P.E. Heimdahl, Devaraj George, and Robert Weber. Specification test coverage adequacy criteria = specification test generation inadequacy criteria? In *HASE '04*, pages 178–186, 2004.

[Hie97] Robert M. Hierons. Testing from a Z specification. *STVR*, 7(1):19–33, 1997.

[Hie04] Rob M. Hierons. Testing from a nondeterministic finite state machine using adaptive state counting. *IEEE Trans. Comput.*, 53(10):1330–1342, 2004.

[HJM05] Thomas A Henzinger, Ranjit Jhala, and Rupak Majumdar. Permissive interfaces. In *ACM SIGSOFT Software Engineering Notes*, volume 30, pages 31–40, 2005.

[HLSU02] H. Hong, I. Lee, O. Sokolsky, and H. Ural. A temporal logic based theory of test coverage and generation. *Tools and Algorithms for the Construction and Analysis of Systems*, pages 151–161, 2002.

[IH14] Laura Inozemtseva and Reid Holmes. Coverage is not strongly correlated with test suite effectiveness. In *ICSE '14*, 2014.

[IX08] K. Inkumsah and Tao Xie. Improving structural testing of object-oriented programs via integrating evolutionary testing and symbolic execution. In *Proceedings of the 2008 23rd IEEE/ACM International Conference on Automated Software Engineering*, ASE '08, pages 297–306, 2008.

[JH11] Yue Jia and Mark Harman. An analysis and survey of the development of mutation testing. *TSE*, 37(5):649–678, 2011.

[JT96] Jan and Tretmans. Conformance testing with labelled transition systems: Implementation relations and test generation. *Computer Networks and ISDN Systems*, 29(1):49–79, 1996.

[JW89] B. Jeng and E. Weyuker. Some observations on partition testing. *ACM SIGSOFT Software Engineering Notes*, 14(8):38–47, nov 1989.

[KKT07] Bogdan Korel, George Koutsogiannakis, and Luay H. Tahat. Model-based test prioritization heuristic methods and their evaluation. In *A-MOST '07*, pages 34–43, 2007.

[LMS07] Lisa Liu, Bertrand Meyer, and Bernd Schoeller. Using contracts and boolean queries to improve the quality of automatic test generation. In *TAP '07*, pages 114–130. 2007.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines-a survey. *Proceedings of the IEEE*, 84(8):1090–1123, 1996.

[MBLDP11] Samar Mouchawrab, Lionel C. Briand, Yvan Labiche, and Massimiliano Di Penta. Assessing, comparing, and combining state machine-based testing and structural testing: A series of experiments. *TSE*, 37(2):161–187, 2011.

[MCLH05] R. E. Miller, D.-L. Chen, D. Lee, and R. Hao. Coping with nondeterminism in network protocol testing. In *TESTCOM '05*, pages 129–145, 2005.

[MIB04] H. Muccini, P. Inverardi, and A. Bertolino. Using software architecture for code testing. *TSE*, 30(3):160–171, 2004.

[MK06] Jeff Magee and Jeff Kramer. *Concurrency - state models and Java programs (2. ed.)*. Wiley, 2006.

[MOK05] Yu-Seung Ma, Jeff Offutt, and Yong Rae Kwon. Mujava: an automated class mutation system: Research articles. *Softw. Test. Verif. Reliab.*, 15(2):97–133, jun 2005.

[MPRS11] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. Autoblacktest: a tool for automatic black-box testing. In *ICSE '11*, pages 1013–1015, 2011.

[MV95] Masaaki Mori and Son T. Vuong. On finite covering of infinite spaces for protocol test selection. In *PSTV '94*, pages 237–251, 1995.

[NA09] Akbar Siami Namin and James H. Andrews. The influence of size and coverage on test suite effectiveness. In *ISSTA '09*, pages 57–68, 2009.

[NABL13] Daniel Di Nardo, Nadia Alshahwan, Lionel C. Briand, and Yvan Labiche. Coverage-based test case prioritisation: An industrial case study. In *ICST '13*, pages 302–311. IEEE, 2013.

[OB88] T.J. Ostrand and M.J. Balcer. The category-partition method for specifying and generating functional tests. *Communications of the ACM*, 31(6):676–686, 1988.

[OLAA03] Jeff Offutt, Shaoying Liu, Aynur Abdurazik, and Paul Ammann. Generating test data from state-based specifications. *STVR*, 13(1):25–53, 2003.

[OP97] A. Jefferson Offutt and Jie Pan. Automatically detecting equivalent mutants and infeasible paths. *Software Testing Verification and Reliability*, 7(3):165–192, 1997.

[PBG04] A. Petrenko, S. Boroday, and R. Groz. Confirming configurations in efsm testing. *IEEE Transactions on Software Engineering*, pages 29–42, 2004.

[PLEB07] Carlos Pacheco, Shuvendu K. Lahiri, Michael D. Ernst, and Thomas Ball. Feedback-directed random test generation. In *Proceedings of the 29th international conference on Software Engineering*, ICSE '07, pages 75–84, 2007.

[PPW+05] A. Pretschner, W. Prenninger, S. Wagner, C. Kühnel, M. Baumgartner, B. Sostawa, R. Zölch, and T. Stauner. One evaluation of model-based testing and its automation. In *ICSE '05*, pages 392–401, 2005.

[PRB09] Charles Pecheur, Franco Raimondi, and Guillaume Brat. A formal analysis of requirements-based testing. In *ISSTA '09*, pages 47–56, 2009.

[RCW08] Matthew J. Rutherford, Antonio Carzaniga, and Alexander L. Wolf. Evaluating test suites and adequacy criteria using simulation-based models of distributed systems. *TSE*, 34:452–470, 2008.

[RHOH98] G. Rothermel, M.J. Harrold, J. Ostrin, and C. Hong. An empirical study of the effects of minimization on the fault detection capabilities of test suites. In *ICSM '98*, 1998.

[Sch99] Steve Schneider. *Concurrent and Real Time Systems: The CSP Approach*. John Wiley & Sons, Inc., New York, NY, USA, 1st edition, 1999.

[Sel03] B. Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, 2003.

[SGA+11] R Sharma, M Gligoric, A Arcuri, G Fraser, and D Marinov. Testing container classes: Random or systematic? In *FASE '11*, pages 262–277, 2011.

[Ton04] Paolo Tonella. Evolutionary testing of classes. In *Proceedings of the 2004 ACM SIGSOFT International Symposium on Software Testing and Analysis*, ISSTA '04, pages 119–128, 2004.

[TXT+11] Suresh Thummalapenta, Tao Xie, Nikolai Tillmann, Jonathan de Halleux, and Zhendong Su. Synthesizing method sequences for high-coverage testing. In *OOPSLA '11*, 2011.

[UL07] M. Utting and B. Legeard. *Practical model-based testing: a tools approach*. Morgan Kaufmann, 2007.

[VCG+08] Margus Veanes, Colin Campbell, Wolfgang Grieskamp, Wolfram Schulte, Nikolai Tillmann, and Lev Nachmanson. Model-based testing of object-oriented reactive systems with Spec explorer. In *Formal Methods and Testing*, volume 4949 of *LNCS*, pages 39–76. 2008.

[VPP06] W. Visser, C.S. Păsăreanu, and R. Pelánek. Test input generation for java containers using state matching. In *ISSTA '06*, pages 37–48, 2006.

[Wei10] Stephan Weissleder. Simulated satisfaction of coverage criteria on uml state machines. In *ICST '10*, pages 117–126, 2010.

[WGS94] E. Weyuker, T. Goradia, and A. Singh. Automatically generating test data from a boolean specification. *Software Engineering, IEEE Transactions on*, 20(5):353–363, 1994.

[WO80] E. J. Weyuker and T. Ostrand. Theory of program testing and the application of revealing subdomains. *IEEE Transactions on Software Engineering*, 6, 1980.

[WSKR06] Kristen R. Walcott, Mary Lou Soffa, Gregory M. Kapfhammer, and Robert S. Roos. Timeaware test suite prioritization. In *ISSTA '06*, pages 1–12, 2006.

[XN04] Tao Xie and David Notkin. Automatic extraction of object-oriented observer abstractions from unit-test executions. In *Formal Methods and Software Engineering*, volume 3308 of *LNCS*, pages 290–305. 2004.

[XXTdH11] Xusheng Xiao, Tao Xie, Nikolai Tillmann, and Jonathan de Halleux. Precise identification of problems for structural test generation. In *Proceedings of the 33rd International Conference on Software Engineering*, ICSE '11, pages 611–620, 2011.

[YH12] S. Yoo and M. Harman. Regression testing minimization, selection and prioritization: A survey. *Softw. Test. Verif. Reliab.*, 22(2):67–120, March 2012.

List of Figures

List of Tables