Pruebas basadas en flujo de datos para programas MapReduce

Jesús Morán, Claudio de la Riva, Javier Tuya

Departamento de Informática, Universidad de Oviedo, Gijón, España
moranjesus@lsi.uniovi.es, {claudio, tuya}@uniovi.es

Resumen. MapReduce es un paradigma de procesamiento masivo de información. Estos programas realizan varias transformaciones de los datos hasta que se obtiene la salida representando la lógica de negocio del programa. En este artículo se elabora una técnica de prueba basada en data flow y que deriva las pruebas a partir de las transformaciones que ocurren en el programa. Se muestran resultados de la ejecución de los casos de prueba derivados de la aplicación de la técnica, los cuales permiten detectar algunos defectos.

Palabras clave: Pruebas de Software, pruebas data flow, MapReduce

1 Introducción

El paradigma *MapReduce* [1] se emplea para el procesamiento de grandes volúmenes de información y se basa en el principio "*divide y venceras*", divide un problema en varios sub-problemas más sencillos. Estos programas manejan pares <*clave, valor*>, donde la clave representa un sub-problema y el valor la información que se necesita para resolverlo.

El resultado final de estos programas proviene de un conjunto de transformaciones de los datos almacenados en los pares <*clave, valor*>. En algunas ocasiones puede suceder que los desarrolladores en lugar de crear programas *MapReduce* con muchas transformaciones, los dividan en varios programas con menos transformaciones para así minimizar los daños de potenciales fallos y no perder todo el procesamiento [2], sacrificando así el rendimiento por temor a fallos. El paradigma *MapReduce* se emplea en sectores críticos como por ejemplo la salud (alineamiento de secuencias de ADN [3]) o la seguridad (análisis de imágenes en balística [4]). Por tanto, la mejora de la calidad de los programas *MapReduce* debería ser una parte importante en el desarrollo, y en particular la realización de pruebas.

Si bien existen algunos enfoques para la prueba de programas *MapReduce*, la mayoría se centran en pruebas de rendimiento [5][6][7] y no en la prueba funcional de este tipo de aplicaciones, que es precisamente el objetivo de este trabajo. En este trabajo se elabora una técnica de prueba (*MRFlow*) con el objetivo de probar la funcionalidad de los programas *MapReduce*, basándose en los criterios de flujo de datos (*data flow* testing) [8]. Dado que la mayor parte de la funcionalidad en este tipo de programas está soportada por las diferentes transformaciones de datos, y que éstas no

son consideradas con suficiente detalle en los criterios *data flow*, aquí se toman como base para derivar los casos de prueba. Para ello, se elabora un grafo del programa sobre el que se extraen varios caminos, y según las transformaciones datos, se prueban éstas bajo diferentes situaciones de prueba.

El resto del trabajo es como sigue: en la sección 2 se resumen los principios del paradigma *MapReduce* y de los criterios de prueba basados en *data flow*, y la sección 3 describe el trabajo relacionado. En la sección 4 se presenta la elaboración de la técnica: elaboración del grafo de flujo de datos (sección 4.1) y la derivación de los casos de prueba (sección 4.2), que se ilustran mediante la aplicación a un caso de estudio. Finalmente, en la sección 5 se presentan las conclusiones.

2 Conceptos Básicos

La técnica de prueba que se propone en este artículo (*MRFlow*) tiene como objetivo derivar pruebas para los programas implementados con el paradigma *MapReduce*. Esta técnica de prueba se basa en los criterios de prueba *data flow* que analizan la evolución del estado de las variables. En la subsección 2.1 se describe el paradigma *MapReduce* junto con un ejemplo, y en la subsección 2.2 los fundamentos de los criterios de prueba *data flow*.

2.1 MapReduce

La programación *MapReduce* divide un problema en varios sub-problemas que se pueden ejecutar en paralelo. Principalmente se implementan dos funciones: *Map* y *Reduce*. La función *Reduce* resuelve el sub-problema y la función *Map* se encarga de hacerle llegar todos los datos que necesita. Para ello, la función *Reduce* recibe un par *<clave, lista(valor)>* donde la clave identifica el sub-problema, y la lista de valores contiene la información necesaria para resolverlo. Por otro lado, la función *Map* recibe la entrada del programa en paralelo y emite pares *<clave, valor>*, siendo la clave el identificador del sub-problema, y conteniendo en el valor parte de la información necesaria para resolverlo.

Considerar a modo de ejemplo un programa en *MapReduce* que contabiliza el número de emails y sms enviados cada año. Este problema se divide en tantos sub-problemas como años, es decir, la clave es el año. La función *Reduce* resuelve cada sub-problema, entonces cuenta los emails y sms enviados para el año almacenado en su clave. La función *Map* se encarga de emitir a cada función *Reduce* los emails y sms que se enviaron ese año en forma de pares *<año, email>* y *<año, sms>*. Tal y como se representa en la figura 1 el problema se divide en tres sub-problemas, uno por cada año. La función *Map* recibe la entrada y según el año de envío del email o del sms, se envía a la función *Reduce* que corresponda. Finalmente la función *Reduce* contiene todos los emails y sms que se enviaron en un año, los cuenta y emite el resultado.

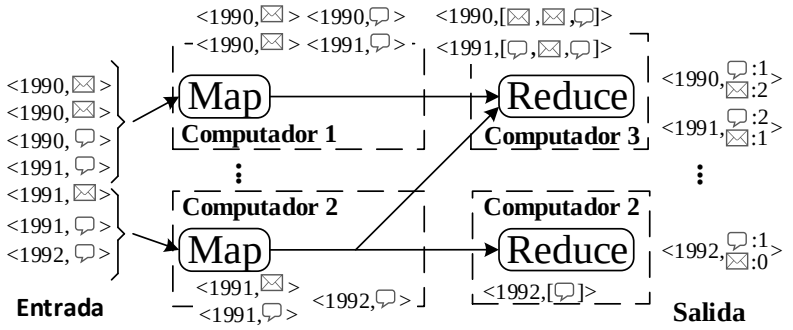

Fig. 1. Programa que cuenta los emails y los sms enviados cada año

Los programas *MapReduce* se suelen emplear para problemas *Big Data* [9], donde se tiene una gran cantidad de datos (Volumen), que se tienen que procesar rápidamente (Velocidad), y puede que aparentemente no tengan modelo de datos como por ejemplo emails, imágenes, etc. (Variedad). Por estas características, los programas *MapReduce* se suelen ejecutar en un framework, destacando *Hadoop* [10] por su impacto en las organizaciones [11].

2.2 Fundamentos de los Criterios de Prueba Data Flow

El objetivo de los criterios de prueba *data flow* es derivar pruebas analizando las variables del programa y teniendo en cuenta sus definiciones. Estos criterios de prueba se basan en la estructura del programa *P* [12]. Se parte de un grafo del flujo de control del programa *G(P)*, en el cual los nodos representan a las instrucciones y los arcos indican la siguiente instrucción. Además del grafo, se establecen las definiciones y usos de cada variable del programa [8]. Cuando en un nodo $n \in N$ se le asigna un valor a una variable $v \in V$, se dice que la variable es definida y se expresa *DEF(v,n)*. Si la variable se utiliza como predicado de una condición (ej: if (v)) se denota mediante *P-USE(v,n)*, y para otros usos de la variable (ej: x = 2*v) se utiliza *C-USE(v,n)*. Por ejemplo, en la instrucción a = b+1 la variable *a* es definida y la variable *b* es utilizada.

Para obtener casos de prueba, Rapps et al. [8] proponen varios criterios que consisten en ejecutar caminos del grafo *G(P)* que cumplan patrones de definiciones y usos de las variables. Por ejemplo, con el criterio "*all-du-paths*" se tienen que probar todos los caminos en los que cada variable $v \in V$ se define $DEF(v,n_1)$ y finalmente se utiliza $USE(v,n_2)$ sin ninguna otra definición intermedia de *v* y sin bucles intermedios.

3 Trabajo Relacionado

En los últimos años se ha progresado en la investigación de técnicas relacionadas con las pruebas [13], pero se han realizado pocos esfuerzos enfocados a paradigmas de procesamiento masivo de datos como *MapReduce* [14], a pesar que se plantean nue-

vos desafíos en las pruebas [15][16]. Entre ellos la validación de información que aparentemente no tiene un modelo de datos, así como medir su calidad [17].

Según una estimación de Kavulya et al. [18] el 3% de los programas *MapReduce* no acaba de ejecutarse, mientras que Ren et al. [2] lo sitúan entre el 1.38% y el 33.11%. Una clasificación de las pruebas en entornos *Big Data* es realizada por Gudipati et al. [19] proponiendo varias comprobaciones y validaciones, entre ellas la "*Validación de proceso MapReduce*". Dentro de este proceso, Camargo et al. [20] y Morán et al. [21] realizan una clasificación de varios defectos funcionales, donde un defecto puede ser probado automáticamente por un framework de ejecución simbólica desarrollado por Csallner et al. [22].

Mattos [23] elabora un algoritmo bacteriológico para la generación de datos de prueba, donde se tiene que diseñar una función específica por cada programa para generar los datos. En las funciones *Map* y *Reduce*, los tipos de datos de entrada y salida deben ser compatibles entre sí, fallando el programa en caso que se reciba otro tipo de dato. Dörre et al. [24] proponen una función que comprueba en tiempo de compilación si las funciones *MapReduce* tienen tipos de datos incompatibles. Puesto que una de las características de estos programas es que se suelen ejecutar sobre computadores que fallan, existen varias herramientas que inyectan fallos en la infraestructura [25][26][27], así como diversas investigaciones [28][29][30][31] que abordan las pruebas de recuperación de los programas ante posibles fallos de la infraestructura.

La técnica de prueba propuesta en este artículo se enmarca dentro de la comprobación "*Validación de proceso MapReduce*" propuesta por Gudipati et al. [19], y se diferencia del resto de trabajos en que las situaciones de prueba son diseñadas sistemáticamente para detectar defectos funcionales a partir del análisis de las transformaciones que se producen en los programas *MapReduce*.

4 Técnica de Prueba MRFlow (MapReduce data Flow)

Estudios realizados en programas del ámbito *MapReduce* han detectado que el 84.5% de los fallos se deben al procesamiento de los datos [32], por lo que basarse en los tipos de transformaciones que se realizan en este procesamiento puede ser un enfoque eficaz para definir pruebas en este tipo de programas.

El resultado de los programas *MapReduce* proviene de una serie de transformaciones de las claves y de los valores de la entrada. Es decir, las claves y los valores de entrada se transforman en unas variables, que a su vez se transforman en otras, y así sucesivamente hasta que se emiten las salidas. Este flujo de transformaciones representa la lógica del programa *MapReduce* ya que su objetivo es transformar las claves y los valores de entrada en la solución. Considerando aspectos específicos de los programas *MapReduce* como el manejo de pares *<clave, valor>*, se propone una técnica de prueba de las transformaciones de datos y que denominamos *MRFlow* (MapReduce data Flow).

Los criterios de prueba *data flow* como por ejemplo "*all-du-paths*" analizan las definiciones y los usos de forma independiente para cada variable, y por tanto no consideran con suficiente detalle las transformaciones que se producen en los programas

MapReduce. La técnica de prueba que se define a continuación analiza las definiciones de las claves/valores y los usos de sus diferentes transformaciones. Puesto que una parte importante de la funcionalidad está en la función *Reduce*, este artículo se centra en las pruebas de la función *Reduce*, pero también se pueden probar de forma similar las funciones *Map*. En la subsección 4.1 se define el grafo *MRFlow* de la función *Reduce*, y en la subsección 4.2 la derivación de los casos de prueba a partir del grafo anterior.

4.1 MRFlow: Creación del Grafo

Cuando una variable del programa se crea a partir de información de la clave y/o de los valores, se dice que la variable es una transformación de la clave y/o de los valores. Todas estas transformaciones forman parte de la funcionalidad del programa y se modelan mediante un grafo *G(Reduce)*, denominado grafo *MRFlow*. Cada nodo del grafo representa una instrucción y los arcos indican la siguiente instrucción a ejecutar. Además, cada nodo se etiqueta con información relativa a la definición de la clave y de los valores, los usos de las transformaciones y la emisión de la salida, tal y como se describe a continuación.

Nodos de uso: Representan una instrucción que utiliza una de las variables proveniente de transformaciones. Esta variable se puede crear a partir de varias transformaciones diferentes y estas pueden contener información de la clave, parte de la clave, un valor, varios valores, y combinaciones de clave y valores.

Dadas una variable *var*, una instrucción *n* y diversas trasformaciones *seq*, se definen los nodos *P-USE-TRANS(var, n, seq)* cuando la variable *var* se utiliza como predicado de una condición en la instrucción *n* y se construye a partir de las transformaciones *seq*; y se definen los nodos *C-USE-TRANS(var, n, seq)* para cualquier otro uso de la variable *var*. La etiqueta *seq* representa la transformación(es) de la variable *var* y utiliza como notación una cadena de combinaciones de las claves y los valores concatenados con conectores lógicos de conjunción $\wedge$ para indicar todas las posibles transformaciones, y con conectores lógicos de disyunción $\vee$ para indicar qué partes de la clave y de los valores están involucrados en cada transformación. Por ejemplo *P-USE-TRANS(var, 6, (clave* $\wedge$ *valor)* $\vee$ *clave)* expresa que la variable *var* se puede formar a partir de una transformación de la clave y del valor, o solamente de la clave, tal y como se representa en el código:

```
0 Reduce(Key key, List values){
1   var;
2   if(values.size()==1)
3     var=key
4   else
5     var=key+values[0];
6   if(isValid(var))...
```

Al analizar estáticamente el código en la instrucción 6, la variable *var* puede provenir de una transformación de la clave (ejecutando 3) o de una transformación de la clave

junto con el valor (ejecutando 5), por tanto el nodo correspondiente a la instrucción 6 se etiqueta con *P-USE-TRANS(var, 6, (clave* $\wedge$ *valor)* $\vee$ *clave)*. Dado que las transformaciones pueden ser entre partes de clave y del valor, estas se representan mediante las siguientes expresiones:

- Transformaciones de clave:
  - [K]: Se emplea toda la clave. Por ejemplo: var = key; o var = key.length();
  - Ki: Se emplea una parte de la clave, que se denota por *i*. En ocasiones se crean claves compuestas por varios elementos. Por ejemplo si se quiere obtener el número de mensajes que se enviaron por cada año a cada persona, se crea una clave conteniendo simultáneamente la persona y el año. Una transformación podría ser: var = getYear(key); que se expresa por Kyear que es la parte de la clave correspondiente al año.
- Transformaciones de valores:
  - [V]: Varios valores de la lista de valores. Por ejemplo: var =values[0]+values[1]
  - V: Un valor de la lista de valores. Por ejemplo: var = values.Next();
- Transformaciones de valores con categorías: La lista de valores que recibe la función *Reduce* puede tener varios tipos de valores que se tratan de forma diferente. Estos tipos se denotan como categorías. Por ejemplo si la función *Reduce* analiza mensajes enviados a clientes, puede ocurrir que esta reciba emails o sms. Cada uno de ellos es una categoría ya que son valores completamente distintos y que se procesan de forma diferente. Identificar estas categorías aporta un valor añadido a las pruebas ya que permite definir transformaciones que involucren valores de esas categorías. Las categorías, cuando existan, se obtienen examinando la naturaleza y los distintos tipos de valores que se pueden recibir en la función *Reduce*.
  - [V:cat]: Varios valores de la lista de valores que pertenecen a la categoría *cat*. Por ejemplo: var = values[0] + values[1]; puede ser una transformación [V], pero si values[0] y values[1] son de la categoría email, entonces es [V:email], como por ejemplo: if(isEmail(values[0]) && isEmail(values[1])) var = values[0] + values[1];
  - V:cat: Un valor de la lista de valores que pertenece a la categoría *cat*. Por ejemplo: if(isEmail(values[0])) var = values[0];

Nodos de definición: Expresan la asignación de nuevo contenido a la clave o a la lista de valores que recibe la función *Reduce*. Dada una variable *var* y una instrucción *n*, se define el nodo *DEF-K(var, n)* cuando en la instrucción *n* se le asigna contenido a la variable *var*, y además *var* es la clave que recibe la función *Reduce*; y se definen los nodos *DEF-V(var, n)* cuando *var* es la lista de valores que recibe la función *Reduce*.

Nodo de emisión: El resultado de las funciones *Reduce* se emite en una instrucción en forma de par <*clave, valor*>. Dadas unas variables $\{k_1,k_2,\ldots k_m\}$, otras variables $\{v_1,v_2,\ldots,v_p\}$ y una instrucción *n*, se definen los nodos $EMIT(\{k_1,k_2,\ldots k_m\}, \{v_1,v_2,\ldots,v_p\}, n)$, cuando en la instrucción *n* se emite un par <*clave, valor*> donde la clave se forma con las variables $k_1,k_2,\ldots k_m$, y el valor con las variables $v_1,v_2,\ldots,v_p$.

En la figura 2 se muestra el grafo *MRFlow* de una función *Reduce* que analiza los mensajes enviados a un cliente (emails o sms), y devuelve la cantidad de emails enviados a cada cliente y en cada año.

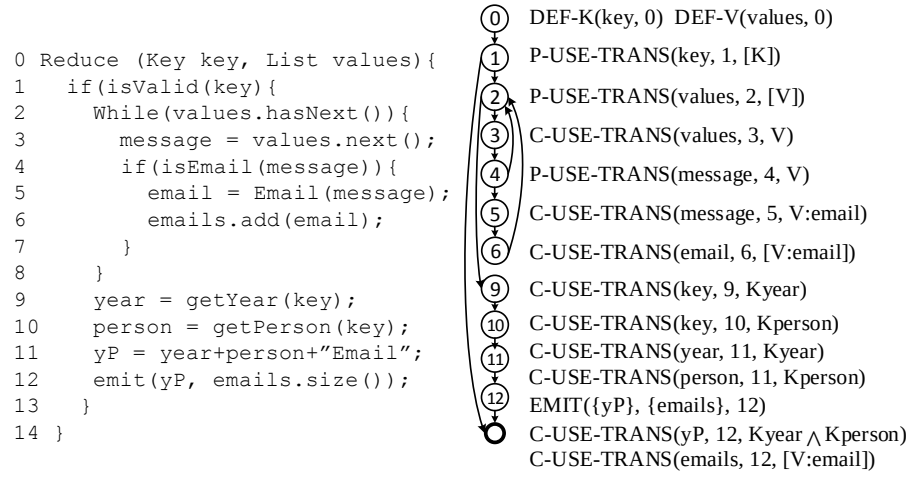

Fig. 2. Grafo MRFlow para programa que cuenta el número de emails por persona y año

La función *Reduce* recibe pares *<[año,persona],lista(mensajes)>*, donde los mensajes pueden ser emails o sms que se envían a un cliente. En el programa se realizan varias transformaciones, por ejemplo *message* en la instrucción 3 se forma con una transformación de un valor *values*. Este a su vez se podría transformar en *email*, entonces significa que el valor pertenece a la categoría *email*. Y así sucesivamente hasta que se tiene la salida del programa. La clave de la salida se forma a partir de las partes de la clave de entrada Kyear y Kperson, es decir, el año y la persona. El valor de la salida se forma mediante varios valores de entrada que pertenecen a la categoría email.

Al añadir estos nodos al grafo *MRFlow*, se pueden obtener caminos bajo prueba que consideren las diferentes transformaciones que ocurren hasta que se genera el resultado final. En la siguiente subsección se detalla la derivación de casos de prueba a partir del grafo *MRFlow*.

4.2 MRFlow: Derivación de los Casos de Prueba

El objetivo de las pruebas *MRFlow* es derivar casos de prueba que analicen las diferentes transformaciones realizadas desde la entrada de la función *Reduce* hasta que se obtiene el resultado. Para ello en el grafo *MRFlow* se extrae un camino desde que se define la clave/valores hasta los usos de sus últimas transformaciones, y se cubre ese camino con varias situaciones de prueba: estando vacío o no el campo de clave/valores, conteniendo información válida o no válida, y emitiendo o no el resultado.

Obtención de caminos bajo prueba: Se extraen del grafo *MRFlow* los caminos sobre los que se ejecutan las diferentes situaciones de prueba. Cada camino bajo prueba

se denomina *tp* (*transformation path*) y su trayecto contiene una transformación. Los caminos *tp* son aquellos que siguiendo las transformaciones van desde la definición de la clave/valores (*DEF-K* o *DEF-V*) hasta cada nodo uso de la variable que contiene la última transformación antes de emitir el resultado (*C-USE-TRANS* o *P-USE-TRANS*). Además, al igual que en otros criterios de prueba *data flow*, si existe *P-USE* (*P-USE-TRANS* en *MRFlow*) se añaden a los caminos bajo prueba (*tp*) todos los siguientes nodos según el grafo *MRFlow*. Entre cada nodo *DEF-K*/*DEF-V* y *P-USE-TRANS*/*C-USE-TRANS* pueden existir varios caminos, incluso infinitos, pero sólo se realizan pruebas sobre uno cualquiera. Además, dependiendo de las transformaciones que tiene el camino *tp*, se tienen que cubrir diferentes situaciones de prueba con diversos datos que existan, estén vacíos, sean válidos, no válidos, que se emita el resultado y que no se emita.

En la figura 3, para el programa de la anterior sección, se representan los caminos *tp* correspondientes a las transformaciones de la lista de valores y de la clave. Se obtienen 13 caminos bajo prueba a partir de las diferentes transformaciones, por ejemplo el camino tp_1: 0→ ...→ 3→ ... → 5→ 6→ ... → 12→ ... tiene una transformación de varios valores de la categoría *email*, mientras que la transformación del camino tp_2: 0→ ... → 3→ 4→ 2→ ... implica un sólo valor. Además, como existen nodos *P-USE-TRANS*, se crea un camino con cada uno de sus siguientes nodos según el grafo *MRFlow*, al igual que se hace en otros criterios *data flow*.

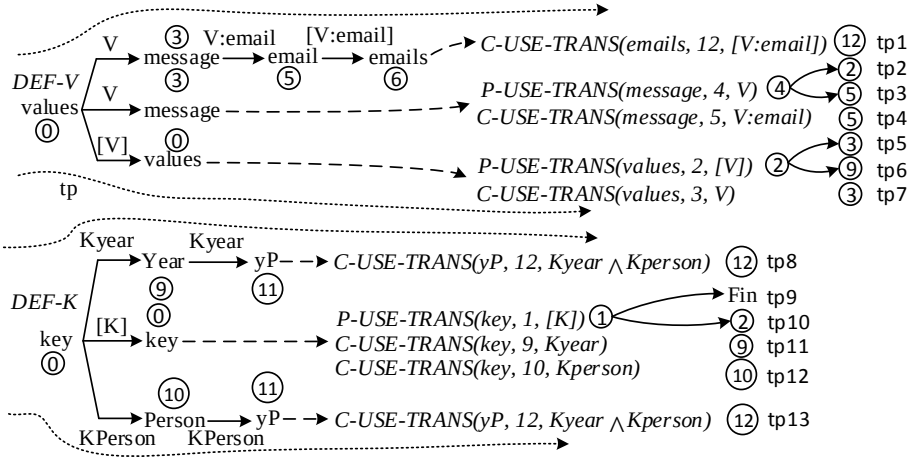

Fig. 3. Transformación y caminos tp para programa que cuenta el número de emails

Obtención de situaciones de prueba: Las situaciones de prueba son las condiciones bajo las que se tiene que probar cada camino *tp*. Por cada uno de estos caminos se abordan diferentes situaciones de prueba. El número y tipo de las situaciones de prueba que se tienen que probar en un camino *tp* depende de la transformación que ocurre en ese camino. Las transformaciones de un camino están reflejadas en la etiqueta *seq* del nodo *P-USE-TRANS(var, n, seq)*/*C-USE-TRANS(var, n, seq)*, y son combinaciones de claves ([K]), parte de la clave (Ki), varios valores ([V] o [V:cat]) y de un valor

(V o V:cat). Las situaciones de prueba que se tienen que probar para cada camino *tp* dependen de la transformación de este, y son las siguientes:

- Existencia de información: La transformación se genera con datos que deben existir o estar vacíos. Dependiendo de la transformación del camino *tp*, puede ocurrir:
  - La transformación de *tp* incluye varios valores sin categoría [V]: Además se tiene que probar por cada una de las categorías [V:cat] que la transformación se genere con varios datos de la categoría *cat* o que se elabore sin datos *cat*. Por ejemplo si se tiene una transformación [V], y los valores pueden ser de categoría [V:email] o [V:sms], entonces se prueba que la transformación se elabore a partir de varios datos de la categoría email y de la categoría sms, y también que se genere sin datos de la categoría email, y sin datos de la categoría sms.
  - La transformación de *tp* incluye toda la clave [K]: Además se tiene que probar que la transformación se genere con toda la clave, o que contenga vacía cada parte de la clave Ki.
- Validez: La transformación se genera con información válida y no válida. Dependiendo de la transformación del camino *tp*, puede ocurrir:
  - La transformación de *tp* incluye varios valores sin categoría [V]: Además se tiene que probar por cada una de las categorías [V:cat] que la transformación se genere con datos de la categoría *cat* válidos o no válidos.
  - La transformación de *tp* incluye toda la clave [K]: Además se tiene que probar que la transformación se genere con toda la clave, o que contenga por cada parte de la clave Ki datos válidos, o no válidos.
- Consecución: Se prueba que el camino *tp* alcance un nodo *EMIT* y que no alcance ningún nodo *EMIT*.

El anterior ejemplo tiene 13 caminos *tp* bajo prueba provenientes de transformaciones de la clave (tp_8-tp_{13}) y de la lista de valores (tp_1-tp_7). En el resto de sección se describen las pruebas para los 7 caminos con transformaciones de la lista de valores, aunque en *MRFlow* se tienen que hacer pruebas cubriendo los 13 caminos *tp*. En cada camino bajo prueba (tp_1-tp_7) se tienen que cubrir las diferentes situaciones de prueba según las transformaciones del camino. Por ejemplo el camino tp_1 tiene una transformación de varios valores de la categoría *email*, [V:email], entonces se tienen que probar las situaciones de prueba donde en la transformación existan varios emails, que no existan los emails (categoría [V:email] vacía), que sean válidos, que no sean válidos, que se llegue a *EMIT* y que no se llegue a *EMIT*. En cambio, el camino tp_5 tiene una transformación de varios valores [V], entonces además de lo anterior se tiene que probar que en la transformación existan o no existan varios sms y que estos sean válidos o no válidos. Así sucesivamente hasta que se obtienen manualmente todas las situaciones de prueba a cubrir en cada camino *tp*, quedando un total de 60 situaciones de prueba entre los 7 caminos, de las cuales 10 no se pueden cubrir. Por ejemplo para el camino tp_1 no se puede cubrir la situación de prueba "no se llega a *EMIT*" porque siempre se llega. Una vez se tienen las situaciones de prueba y los caminos en los que se tienen que cubrir, se crean casos de prueba hasta que se cubran todas las posibles. En este ejemplo de las transformaciones de la lista de valores, el máximo número de situacio-

nes de prueba se pueden cubrir mediante los siguientes casos de prueba, los cuales revelan un defecto por no validar los emails:

- CP1: Su objetivo es cubrir el mayor número de situaciones de prueba.
  - Descripción: Se envían dos emails y dos sms válidos.
  - Entrada: <[Alberto, 1999], [{"email":{"To":"Alberto", "Text": "saludo"}}, {"email":{"To":"Alberto", "Text":"venta"}}, {"sms":{"To":"Alberto", "Text": "contacto"}}, {"sms":{"To":"Alberto", "Text": "saludo"}}]>
  - Camino ejecutado: 0→1→2→3→4→5→6→2→3→4→5→6→2→3→4→2→3→4→2→9→10→11→12→Fin
  - Caminos *tp* cubiertos: tp_1, tp_2, tp_3, tp_4, tp_5, tp_6 y tp_7
  - Salida esperada: <[Alberto, 1999, Email], 2>
- CP2: Su objetivo es cubrir el mayor número de situaciones de prueba no cubiertas por el anterior caso de prueba.
  - Descripción: Se envían dos emails no válidos.
  - Entrada: <[Beatriz, 2000], [{"email":{"To":"Carlos", "Text": "contacto"}}, {"email":{"To:"Beatriz", "Text":"recordatorio"}}]>. El primer email no es válido porque se envía a Carlos y según la clave de la función *Reduce* se están analizando los mensajes enviados a Beatriz. El segundo email no es válido porque le faltan unas comillas en la etiqueta "To".
  - Camino ejecutado: 0→1→2→3→4→5→6→2→3→4→5→6→2→9→10→11→12→Fin
  - Caminos *tp* cubiertos: tp_1, tp_3, tp_4, tp_5, tp_6 y tp_7
  - Salida esperada: <[Beatriz, 2000, Email], 0>.
- CP3: Su objetivo es cubrir el mayor número de situaciones de prueba no cubiertas por los anteriores casos de prueba.
  - Descripción: Se envían dos sms no válidos.
  - Entrada: <[Carlos, 2001], [{"sms": {"To":"Daniela", "Text":"saludo"}}, {"sms": {"Text": "contacto"}}]>. El primer sms no es válido porque se envía a Daniela y el segundo porque no tiene etiqueta "To".
  - Camino ejecutado: 0→1→2→3→4→2→3→4→2→9→10→11→12→Fin
  - Caminos *tp* cubiertos: tp_2, tp_5, tp_6 y tp_7
  - Salida esperada: <[Carlos, 2001, Email], 0>
- CP4: Su objetivo es cubrir las situaciones de prueba que quedan por cubrir.
  - Descripción: Se envían dos emails y dos sms vacíos.
  - Entrada: <[Daniela, 2002],[{"email":""},{"email":""},{"sms":""},{"sms":""}]>
  - Camino ejecutado: 0→1→2→3→4→5→6→2→3→4→5→6→2→3→4→2→3→4→2→9→10→11→12→Fin
  - Caminos *tp* cubiertos: tp_1, tp_2, tp_3, tp_4, tp_5, tp_6 y tp_7
  - Salida esperada: <[Daniela, 2002, Email], 0>

Con los anteriores casos de prueba se cubren 50 de las 60 situaciones de prueba, y dos de ellos fallan por no validar los emails. Algunas situaciones de prueba, en este caso 10, no se pueden ejecutar sobre algunos caminos, al igual que en otros criterios *data flow* [33]. Los casos de prueba CP1 y CP3 se ejecutan correctamente, mientras que CP2 y CP4 generan una excepción no controlada en la instrucción 5 causada por el

mismo defecto. El programa no valida los emails recibidos y al intentar crear un objeto de tipo Email se genera una excepción. De las 60 situaciones de prueba, sólo 3 detectan el defecto: transferencia con emails no válidos en tp_1, o bien transferencia con datos no válidos en tp_3 o en tp_4. En la tabla 1 se representan las situaciones de prueba junto con los casos de prueba que las cubren. El eje horizontal contiene los caminos bajo prueba, el eje vertical las situaciones bajo prueba, y las celdas indican: (×) no se tiene que cubrir esa situación de prueba, (CPi) la situación de prueba se cubre en el camino *tp* con el caso de prueba CPi, y si está en naranja es que revela un defecto, y (⊘) cuando la situación de prueba no se pueda cubrir en ese camino *tp*.

Tabla 1. Situaciones de prueba cubiertas por los casos de prueba

Situaciones de prueba\ tp	tp_1	tp_2	tp_3	tp_4	tp_5	tp_6	tp_7
Transformaciones con datos no vacíos	×	CP1 CP3	CP1 CP2	CP1 CP2	CP1 CP2 CP3 CP4	CP1 CP2 CP3 CP4	CP1 CP2 CP3
Transf. con datos vacíos	×	CP4	CP4	CP4	⊘	⊘	CP4
Transformaciones con emails	CP1 CP2 CP4	×	×	×	CP1 CP2 CP4	CP1 CP2 CP4	×
Transformaciones sin emails	⊘	×	×	×	CP3	CP3	×
Transformaciones con sms	×	×	×	×	CP2	CP2	×
Transformaciones sin sms	×	×	×	×	CP1 CP3 CP4	CP1 CP3 CP4	×
Transf. con datos válidos	×	CP1	CP1	CP1	CP1	CP1	CP1
Transformaciones con datos no válidos	×	CP3 CP4	CP2 CP4	CP2 CP4	CP2 CP3 CP4	CP2 CP3 CP4	CP2 CP3 CP4
Trasnf. con emails válidos	CP1	×	×	×	CP1	CP1	×
Transf. con emails no válidos	CP2 CP4	×	×	×	CP2 CP4	CP2 CP4	×
Transf. con sms válidos	×	×	×	×	CP1	CP1	×
Transf. con sms no válidos	×	×	×	×	CP3 CP4	CP3 CP4	×
Se llega a Emit	CP1 CP2 CP4	CP1 CP3 CP4	CP1 CP2 CP4	CP1 CP2 CP4	CP1 CP2 CP3 CP4	CP1 CP2 CP3 CP4	CP1 CP2 CP3 CP4
No se llega a Emit	⊘	⊘	⊘	⊘	⊘	⊘	⊘

× Situación de prueba no aplicable Condiciones que revelan un defecto

⊘ Situación de prueba que no se puede cubrir

5 Conclusiones

En este trabajo se ha presentado la técnica de prueba *MRFlow* para diseñar las pruebas de los programas *MapReduce*. *MRFlow* se basa en los criterios de prueba *data flow* y considera las características que poseen los programas *MapReduce*. Las situaciones de prueba se derivan mediante un análisis estático de las transformaciones que ocurren en el programa. Esta técnica de prueba se aplica a un programa *MapReduce* y revela la existencia de un defecto por no validar adecuadamente los datos de entrada. Estos resultados preliminares muestran que la técnica propuesta puede ser adecuada para detectar defectos de este tipo, así como otros defectos causados por transformaciones incorrectas de datos en los programas *MapReduce*.

Como trabajo futuro, y con el objetivo de validar la técnica, ésta se aplicará a varios programas reales. Así mismo se abordará la automatización de parte del proceso de pruebas, como puede ser la creación del grafo sobre el que se derivan las pruebas, las situaciones de prueba a cubrir, o la ejecución.

Agradecimientos

Este trabajo ha sido realizado bajo el proyecto de investigación TIN2013-46928-C3-1-R, financiado por el Ministerio de Economía y Competitividad, y fondos FEDER. También ha sido realizado bajo el proyecto GRUPIN14-007, financiado por el Principado de Asturias y fondos FEDER.

Referencias

1. J. Dean and S. Ghemawat. Mapreduce: Simplified data processing on large clusters. In proc. 6th USENIX Sympnosium on Operating Systems Design and Implementation (OSDI), pag-es 137-149. USENIS, Dec. 2004.
2. Kai Ren, YongChul Kwon, Magdalena Balazinska, and Bill Howe. 2013. Hadoop's adolescence: an analysis of Hadoop usage in scientific workloads. Proc. VLDB Endow. 6, 10 (August 2013), 853-864. DOI=10.14778/2536206.2536213
3. Schatz, Michael C.. "CloudBurst: highly sensitive read mapping with MapReduce.." Bioinformatics 25 , no. 11 (2009): 1363-1369
4. Kocakulak, H.; Temizel, T.T., "A Hadoop solution for ballistic image analysis and recognition," High Performance Computing and Simulation (HPCS), 2011 International Conference on , vol., no., pp.836,842, 4-8 July 2011 doi: 10.1109/HPCSim.2011.5999917
5. Kiyoung Kim, Kyungho Jeon, Hyuck Han, Shin G. Kim, Hyungsoo Jung, and Heon Y. Yeom. Mrbench: A benchmark for mapreduce framework. In ICPADS '08: Proceedings of the 2008 14th IEEE International Conference on Parallel and Distributed Systems, pages 11-18, Washington, DC, USA, 2008. IEEE Computer Society.
6. Shengsheng Huang; Jie Huang; Jinquan Dai; Tao Xie; Bo Huang, "The HiBench benchmark suite: Characterization of the MapReduce-based data analysis," Data Engineering Workshops (ICDEW), 2010 IEEE 26th International Conference on , vol., no., pp.41,51, 1-6 March doi: 10.1109/ICDEW.2010.5452747

7. Yanpei Chen; Ganapathi, A.; Griffith, R.; Katz, R., "The Case for Evaluating MapReduce Performance Using Workload Suites," Modeling, Analysis & Simulation of Computer and Telecommunication Systems (MASCOTS), 2011 IEEE 19th International Symposium on , vol., no., pp.390,399, 25-27 July 2011 doi: 10.1109/MASCOTS.2011.12
8. Rapps, S.; Weyuker, E.J., "Selecting Software Test Data Using Data Flow Information," Software Engineering, IEEE Transactions on , vol.SE-11, no.4, pp.367,375, April 1985 doi: 10.1109/TSE.1985.232226
9. Sharma, M.; Hasteer, N.; Tuli, A.; Bansal, A., "Investigating the inclinations of research and practices in Hadoop: A systematic review," Confluence The Next Generation Information Technology Summit (Confluence), 2014 5th International Conference - , vol., no., pp.227,231, 25-26 Sept. 2014 doi: 10.1109/CONFLUENCE.2014.6949381
10. Hadoop: open-source software for reliable, scalable, distributed computing http://hadoop.apache.org/ [Accedido Marzo 2015]
11. Instituciones que utilizan Hadoop con fines educativos o de producción http://wiki.apache.org/hadoop/PoweredBy [Accedido Marzo 2015]
12. IEEE Draft International Standard for Software and Systems Engineering--Software Testing--Part 4: Test Techniques," ISO/IEC/IEEE P29119-4-DISMay2013 , vol., no., pp.1,132, Feb. 21 2014
13. A. Bertolino. Software testing research: Achievements, challenges, dreams. In Proc. of ICSE Future of Software Engineering (FOSE), pages 85–103, May 2007
14. Camargo L. C.; Vergilio S. R. MapReduce program testing: a systematic mapping study. Chilean Computer Science Society (SCCC), 32st International Conference of the Computation, Temuco, Chile, 2013.
15. Nachiyappan. S, Justus. S., "Getting Ready for BigData Testing: A Practitioner's Perception", International Conference on Computational and Networking Technologies, July 2013
16. Akhil Mittal. Trustworthiness of Big Data. International Journal of Computer Applications (0975-887) Volume 80 –Nº.9, October 2013
17. Caballero, I., Serrano, M., Piattini, M. A data quality in use model for Big Data (position paper) (2014) Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics), 8823, pp. 65-74.
18. Kavulya, S.; Tan, J.; Gandhi, R.; Narasimhan, P., "An Analysis of Traces from a Production MapReduce Cluster," Cluster, Cloud and Grid Computing (CCGrid), 2010 10th IEEE/ACM International Conference on , vol., no., pp.94,103, 17-20 May 2010 doi: 10.1109/CCGRID.2010.112
19. Mahesh Gudipati, Shanthi Rao, Naju D. Mohan and Naveen Kumar Gajja. Big Data: Approach to Overcome Quality Challenges. In Big data: Challenges and opportunities. Infosys Labs Briefings. Vol 11 NO 1 2013
20. Camargo L. C.; Vergilio S. R. Classicação de defeitos para programas mapreduce: resultados de um estudo empírico. SAST - 7th Brazilian Workshop on Systematic and Automated Software Testing, 2013
21. Moran, J.; De La Riva, C.; Tuya, J., "MRTree: Functional Testing Based on MapReduce's Execution Behaviour," Future Internet of Things and Cloud (FiCloud), 2014 International Conference on , vol., no., pp.379,384, 27-29 Aug. 2014 doi: 10.1109/FiCloud.2014.67
22. Christoph Csallner, Leonidas Fegaras y Chengkai Li. New Ideas Track: Testing MapReduce-Style Programs. Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering. Pages 504-507
23. Mattos, A. J. Test data generation for testing MapReduce Systems

24. Jens Dörre, Sven Apel, and Christian Lengauer. 2011. Static type checking of Hadoop MapReduce programs. In Proceedings of the second international workshop on MapReduce and its applications (MapReduce '11). ACM, New York, NY, USA, 17-24. DOI=10.1145/1996092.1996096
25. Hadoop Injection Framework. http://wiki.apache.org/hadoop/HowToUseInjectionFramework [Accedido Marzo 2015]
26. AnarchyApe: Fault injection tool for hadoop cluster from yahoo anarchyape. https://github.com/david78k/anarchyape [Accedido Marzo 2015]
27. Chaos Monkey. https://github.com/Netflix/SimianArmy/wiki/Chaos-Monkey [Accedido Marzo 2015]
28. Faraz Faghri, Sobir Bazarbayev, Mark Overholt, Reza Farivar, Roy H. Campbell, and William H. Sanders. 2012. Failure scenario as a service (FSaaS) for Hadoop clusters. In Proceedings of the Workshop on Secure and Dependable Middleware for Cloud Monitoring and Management (SDMCMM '12). ACM, New York, NY, USA, , Article 5 , 6 pages. DOI=10.1145/2405186.2405191
29. Marynowski, J. E. and Pimentel, A. R. (2013). HadoopTest: A Dependability Testing Framework for Hadoop. Technical report, Federal University of Paraná
30. Edson Ramiro Lucas Filho. Implementação de lowertester para sistemas MapReduce utilizando programação orientada a aspectos
31. Pallavi Joshi, Haryadi S. Gunawi, and Koushik Sen. 2011. PREFAIL: a programmable tool for multiple-failure injection. In Proceedings of the 2011 ACM international conference on Object oriented programming systems languages and applications (OOPSLA '11). ACM, New York, NY, USA, 171-188. DOI=10.1145/2048066.2048082
32. Sihan Li, Hucheng Zhou, Haoxiang Lin, Tian Xiao, Haibo Lin, Wei Lin, and Tao Xie. 2013. A characteristic study on failures of production distributed data-parallel programs. In Proceedings of the 2013 International Conference on Software Engineering (ICSE '13). IEEE Press, Piscataway, NJ, USA, 963-972
33. S. R. Vergilio, J. C. Maldonado, and M. Jino, “Infeasible paths in the context of data flow based testing criteria: Identification, classification and prediction,” Journal of the Brazilian Computer Society, vol. 12, no. 1, pp. 73–88, 2006