Classificação de Defeitos para Programas MapReduce: Resultados de um Estudo Empírico

Luiz C. Camargo, Silvia R. Vergilio

[1]DInf- Universidade Federal do Paraná (UFPR)
C.P.19.081 - CEP 85902-490, Curitiba, PR - Brasil

{lccamargo,silvia}@inf.ufpr.br

Resumo. *Existem trabalhos relacionados à qualidade e confiabilidade dos ambientes que executam programas MapReduce (MR), entretanto poucos ainda são os que se dedicam ao teste destes programas. Uma etapa importante para embasar futuros trabalhos neste novo tema de pesquisa é a identificação dos principais defeitos encontrados em programas MR, e de uma classificação de defeitos para nortear diferentes tarefas da atividade de teste. Descrever resultados desta etapa é o objetivo do presente trabalho. Aplicando uma metodologia baseada na manipulação de código fonte e na documentação das APIs MR-MPI e Hadoop, este artigo introduz sete classes de defeitos genéricas e apresenta exemplos de defeitos em cada classe. A classificação proposta é útil para auxiliar a geração de dados de teste, a proposição de critérios de teste e implementação de ferramentas.*

Abstract. *In the literature we find works related to the quality and reliability of the platforms that execute MapReduce (MR) programs. However, few works address the test of such programs. To give research directions in this new field, an important step is the identification of faults that can be present in MR programs, as well as, the definition of a fault classification to serve as basis for different testing tasks. This paper presents results of such step. By using a methodology based on analysis of source code, and of MR-MPI and Hadoop APIs documentation, this paper introduces seven generic fault classes and presents examples of faults from each class. The proposed classification is useful to test data generation, test criteria definiton and implementation tools.*

1. Introdução

O modelo de programação MapReduce (MR) tem sido amplamente utilizado como um padrão para processamento e desenvolvimento de aplicações paralelas e distribuídas envolvendo grandes volumes de dados, tais como: *cloud computing*, *grid computing* e *clusters* [Miner and Shook 2012]. Para facilitar o desenvolvimento dessas aplicações algumas APIs (*Application Programming Interfaces*) para o modelo MR são utilizadas. O MR Hadoop [Apache 2013] e a biblioteca MR-MPI [Plimpton and Devine 2009] são exemplos de tais APIs. O MR Hadoop é um *framework* implementado pela linguagem de programação Java sob o sistema de arquivo distribuído HDFS (*Hadoop Distributed File System*). A biblioteca MR-MPI foi desenvolvida para as linguagens de programação C++, C e Python sob o modelo padrão de passagem de mensagens MPI (*Message Passing Interface*), adotado para desenvolvimento de aplicações paralelas [Snir et al. 1996].

Na literatura, alguns trabalhos tratam da qualidade associada aos ambientes que executam programas MR com o objetivo de garantir confiabilidade e disponibilidade [Wang and Wei 2011, Tang et al. 2010]. Isto permite o desenvolvimento de tal maneira que os esforços estejam concentrados nas operações *map* e *reduce* da aplicação sem que haja preocupações com o controle de execução paralela das operações MR. Entretanto, o desenvolvimento dessas aplicações está sujeito a erros, e ainda são poucos os trabalhos que se preocupam com o teste de programas escritos no estilo MR. Dentre estes destacam-se [Dorre et al. 2011, Csallner et al. 2011].

O trabalho de Dorre, Apel e Lengauger (2011) emprega a verificação estática de códigos de programas Hadoop por meio de uma ferramenta, cuja finalidade é detectar defeitos de incompatibilidade de tipos de dados entre os pares de chave/valor nas operações *map*, *combiner* e *reduce*. Já o trabalho de Csallner, Fegaras e Li (2011) utiliza a técnica de execução simbólica com o objetivo de derivar casos de teste para programas MR Hadoop. Os casos de teste são executados para revelar defeitos em programas nos quais a ordem dos resultados produzidos pela operação *map* ou pela operação *combiner* é relevante para o processamento correto da operação *reduce*. Estes trabalhos são úteis e contribuem para a qualidade de programas MR Hadoop, mas observa-se a inexistência de trabalhos que visam a aplicação de técnicas e critérios de teste neste contexto. Para isto, considera-se que é necessário, por se tratar de um modelo novo, que sejam identificados os principais erros cometidos neste tipo de desenvolvimento e possíveis defeitos aos quais os programas MR estão sujeitos. Os trabalhos encontrados e mencionados acima não incluem uma classificação de defeitos para os programas escritos no estilo MR.

A classificação de defeitos é uma maneira de organizar os vários tipos de defeitos presentes em um tipo de software, os quais muitas vezes são organizados em categorias. A classificação pode ser simples (severidade baixa, meia e alta), ou uma classificação mais detalhada com descrições de cada tipo defeito. O entendimento sobre os tipos de defeitos favorece o projeto de casos de teste, a implementação de ferramentas para auxiliar no processo de teste, contribui para a criação de métricas de teste de software, dentre outros benefícios [Thung et al. 2012]. O teste de programas MR é uma área emergente de pesquisa que pode avançar com esses benefícios.

Considerando tal fato, este trabalho tem como objetivo propor uma classificação de defeitos para programas escritos no modelo MR. São introduzidas sete classes de defeitos genéricas para o modelo MR, derivadas com o emprego de uma metodologia inspirada no trabalho de Nakamura et al. (2006). Os defeitos são agrupados por similaridade e também por grau de dificuldade de serem revelados. Exemplos de cada classe de defeitos são apresentados, mostrando como as classes podem ser utilizadas para programas escritos em MR-MPI (C++) e Hadoop (Java).

O artigo está organizado como segue. Na Seção 2, encontra-se uma breve descrição do modelo MR. Na Seção 3, alguns trabalhos relacionados à classificação de defeitos são descritos seguidos da metodologia para a classificação de defeitos adotada no trabalho. Na Seção 4, é descrito como a metodologia foi aplicada. Na Seção 5, são apresentados os resultados. Em seguida, na Seção 6, encontram-se as conclusões.

2. O Modelo MapReduce (MR)

O MR é um modelo de programação paralela para processamento de grandes quantidades de dados, proposto por [Dean and Ghemawat 2004], basicamente a computação nesse modelo dá-se em duas fases pelas funções *map* e *reduce*, respectivamente executadas nos *reducer* e *mapper nodes*, os quais são organizados seguindo um modelo de comunicação, por exemplo, o modelo *master/slave* [Plimpton and Devine 2011].

O modelo MR pode ser executado em diferentes plataformas e em ambientes distintos, isto é, as aplicações MR podem ser implementadas para a GPU ou núcleos do processador de uma máquina, para um *cluster*, para uma *cloud* ou para uma *grid*. Na maioria dos casos, as implementações das operações MR seguem a arquitetura *master/slave*, na qual as fases de mapeamento e de redução são executadas [Plimpton and Devine 2011, Tang et al. 2010, Dean and Ghemawat 2008].

Fase de mapeamento:

- a função *map*, escrita pelo desenvolvedor, recebe dados estáticos do *master node*, os quais são fragmentados e distribuídos sobre os *mapper nodes*;
- em seguida, a função *map* processa cada fragmento. O resultado do processamento é uma lista contendo pares de chave (k) e valor (v), *list(k,v)*;
- a lista obtida pelo passo anterior é reorganizada pelo *mapper node* e transformada em uma nova lista intermediária *(intermediate list)* contendo pares de chave/valores, nos quais cada chave aparece somente uma vez com os respectivos valores concatenados *(k, list(v))*;
- as listas intermediárias, produzidas pelos *mapper nodes*, podem ser manipuladas pela função *combiner* (também escrita pelo desenvolvedor) antes de serem enviadas à função *reduce* (fase redução). A função *combiner* é utilizada para um pré-processamento da fase de redução e provê ganho de desempenho para alguns domínios de aplicações, porém ela não é obrigatória no processamento MR.

Fase de redução:

- a função *reduce*, também escrita pelo usuário, recebe a lista intermediária no *reducer node*, no qual a função é executada para todos os valores de *(k,list(v)* para uma chave específica que resulta em uma lista de valores *list(v)*;
- quando os *reducer nodes* concluírem o processamento, os resultados são montados e enviados de volta ao *master node*.

3. Metodologia para Classificação de Defeitos

O processo de classificação de defeitos é favorecido pela adoção de uma metodologia. Algumas metodologias são descritas em estudos empíricos encontrados na literatura [Chillarege et al. 1992, Li et al. 2010, Nakamura et al. 2006, Zheng et al. 2006] e assim como o padrão IEEE [IEEE 1998] têm sido utilizadas para classificação de defeitos. A abordagem *Orthogonal Defect Classification* [Chillarege et al. 1992] (ODC) é uma das mais conhecidas. Na ODC a classificação de defeitos é baseada no que se sabe a respeito do defeito, usando atributos como o tipo de defeito ou gatilho (condição do defeito aparecer), e não na opinião de como o defeito foi produzido. O atributo tipo de defeito é utilizado para demonstrar o relacionamento e os efeitos entre os tipos de defeitos encontrados em diferentes fases do processo de desenvolvimento de software.

Em [Nakamura et al. 2006] é proposta uma metodologia para classificação de defeitos baseada na inspeção de código fonte, a qual considera o histórico de versões de código para efetuar a busca por defeitos. A metodologia possui três processos: processo de suporte (atividades de apoio à metodologia); processo de análise (atividades para escolha e análise de código fonte); e processo de verificação (atividades para verificar a consistência da análise). Algumas destas metodologias/classificações possuem passos que só podem ser aplicados a contextos específicos, por isto, neste trabalho foi adotada a metodologia da Figura 1, adaptada da proposta em [Nakamura et al. 2006]. Essa adaptação foi necessária por não ter sido encontrado para este trabalho um repositório com um histórico de versões para programas MR.

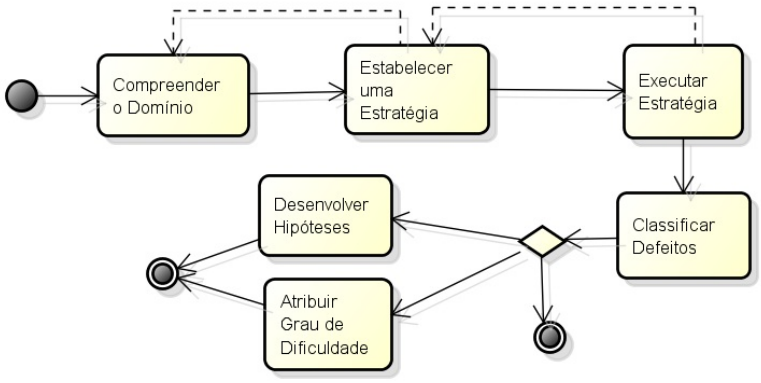

Figura 1. Metodologia para Classificação de Defeitos

A primeira atividade inclui o entendimento do domínio no qual a classificação de defeitos será realizada. Isto pode ser efetuado mediante análise de documentação do domínio, especificação do código fonte ou do próprio código fonte. A segunda tem como objetivo estabelecer uma estratégia, um conjunto de passos, para a identificação de defeitos e consequente classificação. Após o planejamento, os passos devem ser executados, ou seja, a estratégia é aplicada e os defeitos identificados devem ser registrados.

Na quarta atividade, o classificador deve analisar cada defeito identificado a fim de encontrar atributos que levem à definição de classes de defeitos. A definição deve considerar o propósito da classificação e as seguintes propriedades básicas: 1) *ortogonalidade*: cada defeito deve pertencer a uma única classe; 2) *completeza*: todos os defeitos de interesse devem ser classificados em alguma das classes; e 3) *consistência*: diferentes analistas devem classificar um defeito em uma mesma classe. O classificador deve observar a similaridade entre os defeitos e agrupá-los considerando as características comuns entre eles. Cada classe de defeito deve ter uma sigla única para facilitar a sua identificação e uma descrição. Assim como os defeitos, as classes de defeitos devem ser registradas em um template.

As duas próximas atividades são opcionais. Se desejado, hipóteses podem ser geradas para capturar as características dos defeitos em alto nível de abstração, podendo-se gerar perguntas tais como: *Quais classes de defeitos são mais presentes*? *Quão árduo é revelar defeitos de uma determinada classe*? *Como os defeitos de um determinada classe podem ser detectados*? *Qual ou quais critérios de teste podem ser adotados para revelar defeitos para as classes de defeitos identificadas*? Outra atividade que pode ser

realizada é avaliar o grau de dificuldade de cada classe, associado ao esforço exigido para revelar defeitos que pertençam à classe avaliada. Uma possível maneira de executar essa atividade é analisar as características de um conjunto de defeitos de uma determinada classe, e desenvolver casos de teste na tentativa de revelar os defeitos desse conjunto.

4. Classificando Defeitos em Programas MR

A seguir apresenta-se uma descrição das atividades da metodologia descrita na seção anterior aplicadas para a classificação de defeitos em programas MR.

Compreendendo o domínio: o domínio de busca de defeitos neste trabalho inclui programas em C++ derivados da biblioteca MR-MPI [Plimpton and Devine 2009] e programas em Java escritos sob o *framework* MR Hadoop [Apache 2013, Miner and Shook 2012].

Para entender o funcionamento e o uso da biblioteca MR-MPI, foi utilizada a documentação técnica da biblioteca[1] e a execução de programas implementados em C++ (com chamadas procedimentais) sob a biblioteca MR-MPI. A mesma maneira foi adotada para compreender o funcionamento do *framework* MR Hadoop. Além da documentação técnica do framework[2] [White 2012], também foram utilizados programas MR escritos em Java e executados sob *framework* Hadoop (*standalone mode*). O trabalho de Miner e Shook (2012) foi relevante para o entendimento sobre os padrões e boas práticas de implementação do modelo de programação MR.

Estabelecendo uma estratégia: nesta atividade foram estabelecidos os passos da estratégia para identificar defeitos em programas MR, os quais devem ser executados na seguinte ordem:

1. *Analisar primitivas e classes*: analisar as primitivas da biblioteca MR-MPI e também as principais classes e interfaces do framework MR Hadoop utilizadas no desenvolvimento de programas MR. A análise deve fornecer informações sobre o uso adequado das classes e primitivas favorecendo a execução dos próximos passos desta atividade;
2. *Inspecionar código fonte*: ler códigos fontes dos programas `wordfreq.cpp` (MR-MPI) e `WordCount.java` (Hadoop) descritos em [Apache 2013, Miner and Shook 2012], com a perspectiva de encontrar defeitos;
3. *Desenvolver programas*: desenvolver diferentes programas utilizando a biblioteca MR-MPI como também o *framework* MR Hadoop. Isto para permitir ao classificador uma perspectiva de desenvolvedor;
4. *Identificar probabilidade de enganos*: analisar as características de implementações das primitivas MR-MPI e das classes Hadoop com o objetivo de identificar quais primitivas e classes, ao serem implementadas, apresentam complexidades elevando a probabilidade de enganos pelo programador. Este passo deve incluir os programas dos passos 2 e 3;
5. *Inserir defeitos*: alterar os programas utilizados e implementados nas fases anteriores com a finalidade de inserir defeitos. As alterações devem considerar características do modelo de programação MR, como o uso de chamadas a funções

[1] *http://www.sandia.gov/ sjplimp/mapreduce.html*
[2] *http://hadoop.apache.org/docs/current/api/org/apache/hadoop/*

e métodos, uso de parâmetros do ambiente, assinaturas de métodos e funções, passagem de dados entre as operações *map* e *reduce*, dentre outras.

6. *Documentar defeitos*: documentar os defeitos obtidos em um template para posterior classificação, contendo no mínimo a origem do defeito, o exemplo do defeito e uma breve descrição de como esse defeito pode ser ativado/revelado.

Executando a estratégia: os passos da estratégia estabelecidos na atividade anterior foram executados obedecendo a ordem e o propósito dos mesmos. Entretanto, vale ressaltar que na execução do passo 3, foram implementados quatro domínios diferentes de programas [Miner and Shook 2012]; um programa para classificar e contar ocorrência de palavras; um programa para calcular a média de valores de cada termo; um programa para calcular a média de termos com uso de *combiner* e por fim um programa para calcular a menor distância entre nós de um grafo. Esses domínios foram implementados respectivamente com o uso da biblioteca MR-MPI e do *framework* MR Hadoop. A execução dos passos da estratégia resultou na identificação dos defeitos.

Classificando os defeitos: a partir da documentação dos defeitos obtidos na atividade anterior, classes de defeitos foram definidas para os defeitos dos programas em MR-MPI e Hadoop. Para isso foram feitas três observações: foram identificadas classes de defeitos equivalentes, isto é, classes que classificam os mesmos defeitos encontrados em programas MR-MPI como também em programas MR Hadoop; definição de uma nomenclatura única para as classes de defeitos de programas MR-MPI e para as classes de defeitos de programas MR Hadoop; adoção do termo módulo para referir-se tanto a funções (MR-MPI C++) como a métodos (Hadoop java).

Atribuindo um grau de dificuldade às classes: nesta atividade, graus de dificuldade foram estabelecidos de acordo com as características dos defeitos e também por meio da criação e execução de um conjunto de casos de teste para o contexto MR. Os graus de dificuldades em revelar defeitos são descritos pela Tabela 1.

Tabela 1: **Graus de Dificuldade**

Dificuldade	**Tipo**	**Característica da Dificuldade**
D1	Simples	*Para esta dificuldade, um simples caso de teste revela o defeito. Por exemplo, meramente executar o programa.*
D2	Médio	*Ao contrário da Dificuldade D1, a Dificuldade D2, requer um caso de teste mais elaborado, por exemplo: casos de teste com entrada de dados que exercitem ponto(s) específico(s) do programa.*
D3	Difícil	*Esta dificuldade inclui a Dificuldade 2 (D2) com a parametrização de ambiente e/ou a inclusão de um ou mais artifícios. Este tipo de defeito exigirá casos de teste que inclua a dificuldade 2 e exercitem parâmetros de ambiente e/ou o uso de artifícios adicionais.*
D4	Complexo	*A Dificuldade D4 inclui a Dificuldade 3 (D3) e trata defeitos de sincronismo. Também inclui neste tipo de dificuldade, o tratamento do não determinismo, isto é, testar programas de forma determinística. Nesta dificuldade, os casos de teste deverão exercitar o sincronismo entre processos e utilizar alguma técnica para tratar o não determinismo.*

5. Resultados

Foram identificadas sete classes de defeitos, apresentadas com as respectivas siglas, descrições e graus de dificuldade na Tabela 2.

Tabela 2: **Classes de Defeitos - MapReduce**

Tipo Defeito	SIGLA	Descrição	Dific.
Atribuição Incorreta de Argumentos	**ATR_INC_ARG**	*Defeito decorrente da atribuição incorreta de argumentos. Isto é, a atribuição de um ou mais argumentos incorretos para as assinaturas dos módulos. Também pode ser considerado um defeito de atribuição, defeito causado pela propriedade inválida atribuída ao argumento do módulo.*	D1
Uso de Módulo Incorreto	**USO_MOD_INC**	*A escolha inadequada de um ou mais módulos para a implementação de operações de mapeamento e/ou de redução pode levar tais operações a se comportarem de modo inesperado. Este tipo de defeito é potencializado devido às alternativas de módulos encontradas nas APIs para o desenvolvimento de programas MapReduce.*	D2
Ordem de Chamada Incorreta	**ORD_CHA_INC**	*O processamento MapReduce exige uma sincronização na ordem de chamadas de classes ou módulos. Principalmente para as execuções das operações de mapeamento, tratamento de dados intermediários e redução. Uma chamada fora de ordem contida no programa pode levar o programa a falhar.*	D2
Chamada Ausente de Módulo	**CHA_AUS_MOD**	*Defeito decorrente da ausência de chamada a um ou mais módulos necessários ao processamento MapReduce. Isto é, o programa não contempla as chamadas indispensáveis à execução correta do programa.*	D1
Uso de Assinatura Incorreta	**USO_ASS_INC**	*Defeito relacionado ao uso incorreto de assinatura para implementação de um módulo, uma vez que há módulos com mais de uma alternativa de assinatura. Ao utilizar tais módulos o programador pode empregar uma assinatura não adequada ao domínio do programa.*	D2
Inicialização Incorreta de Parâmetro	**INI_INC_PAR**	*Este tipo de defeito pode ser visto como um sub-tipo de **ATR_INC_ARG** (falta de ortogonalidade). Programas MR exigem a configuração de um ambiente (distribuído) o qual é inicializado para execução das operações map e reduce. Porém, a parametrização padrão do ambiente pode ser alterada pelo programa. Neste caso, entende-se que os defeitos provenientes da alteração incorreta desses parâmetros devem ser tratados em separado, uma vez que esses parâmetros lidam com recursos do ambiente (memória, acesso a disco, nós, entre outros recursos) que são empregados no processamento MR.*	D4
Manipulação Incorreta de Dados Intermediários	**MAN_INC_INT**	*Em geral, o processamento MR permite o tratamento de dados intermediários, isto é, os dados produzidos pela operação de mapeamento podem ser tratados para otimizar a operação de redução. No entanto, ao programar o tratamento de dados intermediários (combiner), o desenvolvedor pode mudar o tipo de dado e passar chaves/valores incompatíveis para operação de redução e comprometer o resultado do processamento.*	D3

Vale a pena ressaltar que estas classes são genéricas e podem ser usadas/instanciadas tanto para programas MR-MPI como Hadoop. A título de ilustração, considere as Tabelas 3 e 4, que mostram exemplos de defeito para cada uma das classes, respectivamente,

```
44 int main(int narg, char **args)
45 {
46    MPI_Init(&narg,&args);
47    int me,nprocs;
48 ...
56 MapReduce *mr = new MapReduce
   (MPI_COMM_WORLD);
57 mr->mapstyle = 5;
58
59 mr->map(narg-1,&args[1],0,0,1,
   fileread,NULL);
60 mr->aggregate(NULL);
61 mr->convert();
62
63 mr->reduce(sum, NULL);
64 mr->collate(NULL);
65 mr->map(mr,output,NULL);
66 delete mr;
67 MPI_Finalize();
```

```
68 }
69 void fileread(int itask, char *fname, KeyValue
   *kv, void *ptr)
70 {
71  ...
101 void combiner(char *key, int keybytes, char *multivalue,
102 int nvalues, int *valuebytes, KeyValue *kv, void *ptr)
103 {
104 ...
106    for ( int i = 0; i < nvalues; i++ )
107    {
108    soma +=  *reinterpret_cast<int*>(multivalue
       + *sizeof(int));
109    cnt += 1;
110    }
111 IntPair saida(soma, cnt);
112 kv->add(100,key,keybytes,POINTER_TO_CHAR_PTR(&saida),
    sizeof(IntPair) );
113 }
```

Figura 2. Programa Mediacom.cpp com Defeitos

para os programas MR-MPI `mediacom.cpp` (Figura 2) e `MediaErMain.java` MR Hadoop (Figura 3). Apenas alguns trechos dos programas são apresentados. Estes trechos possuem defeitos descritos nas respectivas tabelas.

Tabela 3: **Exemplos de Classificação de Defeitos MR-MPI**

Linha	**Classe**	**Considerações**
linha 57	**INI_INC_PAR**	*O parâmetro `mapstyle` define como a operação de mapeamento será designada aos processos. Valores aceitáveis para esse parâmetro são: 0, 1 ou 2, e não 5.*
linha 59	**ATR_INC_ARG**	*Os valores atribuídos para o quarto e quinto argumentos da chamada estão incorretos, devem ser iguais a 1.*
linhas 60 e 61	**USO_MOD_INC**	*Os módulos contidos nas chamadas das linhas 60 e 61 devem ser substituídos pelo módulo `mr->collate(NULL)`, uma vez que os módulos utilizados não são adequados ao domínio do programa.*
linhas 63 e 64	**ORD_CHA_INC**	*A ordem das chamadas contidas nessas linhas estão incorretas, visto que é necessário gerar chaves multivaloradas `collate`, antes de invocar o módulo `reduce`.*
linhas 64 e 65	**CHA_AUS_MOD**	*Entre as linhas 64 e 65 deveria ter uma chamada ao módulo `MPI_Barrier(MPI_COMM_WORLD)` visto que, antes de emitir o resultado do processamento MR, a sincronização entre processos é necessária.*
linha 112	**USO_ASS_INC**	*A assinatura utilizada para o módulo na linha 112 está incorreta para o domínio do programa. A assinatura deve ser `kv->add(key, keybytes, POINTER_TO_CHAR_PTR(&saida), sizeof(IntPair))`*
linhas 108 e 109	**MAN_INC_INT**	*O código fonte contido nas linhas 108 e 109 interfere na propriedade dos valores intermediários. Segue o código adequado ao propósito do programa: `IntPair interm = *POINTER_TO_INTPAIR_PTR(multivalue + i*sizeof(IntPair)); soma += interm.first; cnt += interm.second;`.*

```
1 import org.apache.hadoop.io.IntWritable;
5 ...
6 public class MediaErMapper extends
    Mapper<IntWritable,Text,Text,IntArrayWritable>{
8 ...
11 public class MediaErMain {
12      public static final String
    DEFAULT_MAPRED_TASK_JAVA_OPTS = "-Xmx002m";
20 public static Job getInstance(Configuration conf)
   throws IOException { ...
27 System.exit(job.waitForCompletion(true) ? 0 : 1);
28  job.setMapperClass(MediaErMapper.class);
29 ...
33 FileOutputFormat.getOutputPath(job);
34
35job.setReducerClass(MediaErReduce.class);
40 public class MediaErCombine extends
   Reducer<Text, IntArrayWritable, Text,
   IntArrayWritable> {
41 ...
47 for (IntArrayWritable array : arg1) {
48   sum += ((IntWritable)array.get()[0]).get();
49   count += 1;
50 }
```

Figura 3. Programa MediaErMain.java com Defeitos

Tabela 4: **Exemplos da Classificação de Defeitos - Hadoop**

Linha	Classe	Considerações
linha 12	**INI_INC_PAR**	*O parâmetro `DEFAULT_MAPRED_TASK_JAVA_OPTS` está com uma atribuição inadequada e pode inviabilizar a execução de operações de mapeamento. Um valor adequado para o parâmetro: `-Xmx200m`.*
linha 6	**ATR_INC_ARG**	*O primeiro argumento usado pela classe `MediaErMapper` está incorreto, esse argumento deve ser `LongWritable`.*
linha 33	**USO_MOD_INC**	*O módulo utilizado na linha 33 não é o adequado para o programa. O módulo correto é `FileOutputFormat.setOutputPath(job, new Path(args[1]));`.*
linha 27	**ORD_CHA_INC**	*A chamada contida na linha 27 deve estar na linha 38, uma vez que é necessário obter antes os recursos para a execução do `job` para depois invocar o módulo que identifica a finalização do `job`.*
linha 34	**CHA_AUS_MOD**	*Na linha 34, está faltando uma chamada, a qual deve ser chamada como: `job.setCombinerClass(MediaEr Combine.class)`.*
linha 20	**USO_ASS_INC**	*A assinatura utilizada para o módulo na linha 20 está incorreta. A assinatura para o domínio do programa: `public static Job getInstance (Configuration conf,String jobName) throws IOException`.*
linha 49	**MAN_INC_INT**	*O código fonte contido na linha 49 altera a propriedade dos valores intermediários das chaves. O código adequado para a linha 49 é `count += ((IntWritable)array.get()[1]).get();`.*

6. Conclusões

Neste trabalho, por meio de uma metodologia baseada na manipulação de código fonte e documentação relativa às APIs MR-MPI e Hadoop, buscou-se a compreensão e a classificação de defeitos para programas escritos no modelo de programação MR. Foram introduzidas e exemplificadas sete classes de defeitos, associadas aos seus respectivos graus de dificuldade. Estas classes podem ser utilizadas com diferentes propósitos no contexto de teste de programas MR, tais como: projeto e elaboração de casos de teste, proposição de critérios de teste, implementação de ferramentas de suporte, estabelecimento de métricas de teste, estimativa da confiabilidade de software, e avaliação de um processo de teste de software específico para este contexto.

Vale a pena ressaltar que as classes de defeitos foram obtidas de maneira empírica. Algumas ameaças à validade dos resultados estão relacionadas à metodologia adotada, ao número restrito de programas utilizados e as questões subjetivas envolvidas na tarefa de classificação. Por isto esta classificação deverá ser validada em trabalhos futuros. Entre-

tanto as classes propostas constituem uma classificação preliminar para este modelo de programa e são bastante úteis devido a inexistência de trabalhos similares reportados na literatura, podendo impulsionar e fundamentar trabalhos futuros sobre o teste de programas MR.

Referências

Apache, S. F. (2013). Apache Hadoop API. Technical report, Apache Software Foundation.

Chillarege, R., Bhandari, I. S., Chaar, J. K., Halliday, M. J., Moebus, D. S., Ray, B. K., and Wong, M.-Y. (1992). Orthogonal Defect Classification-A Concept for In-Process Measurements. *IEEE Trans. on Soft. Engin.*, 18(11):943–956.

Csallner, C., Fegaras, L., and Li, C. (2011). New Ideas Track: Testing MapReduce-Style Programs. *ACM SIGSOFT Symposium- ESEC/FSE11. Szeged, Hungary*, pages 17–24.

Dean, J. and Ghemawat, S. (2004). MapReduce: Simplified Data Processing on Large Clusters. *OSDI04: Sympos. Operating System Design and Implem.*, (24):137–150.

Dean, J. and Ghemawat, S. (2008). MapReduce: Simplified Data Processing on Large Clusters. *Communications of the ACM*, 51(7):107–113.

Dorre, J., APel, S., and Lengauer, C. (2011). Static Type Checking of Hadoop MapReduce Programs. *ACM - MapReduce'11, San Jose, California, USA*, pages 17–24.

IEEE, C. S. (1998). 829-1998-IEEE Standard for Software Test Documentation. [Online, accessed 10-Maio-2013].

Li, N., Li, Z., and Sun, X. (2010). Classification of Software Defect Detected by Black-box Testing: An Empirical Study. In *WRI World Congress on Softw. Engin.*, pages 234–240.

Miner, D. and Shook, A. (2012). *MapReduce Design Patterns*, volume 1. O'Relly Midia, Inc, www.it-ebooks.info, 1 ed. edition.

Nakamura, T., Hochstein, L., and Basili, R. V. (2006). Identifying Domain-Specific Defect Classes Using Inspections and Change History. *ACM, ISESE, Rio de Janeiro, Brazil*, pages 346–355.

Plimpton, J. S. and Devine, D. K. (2011). MapReduce in MPI for Large-Scale graph algorithms. *Elsevier, Parallel Computing*, 37:610–632.

Plimpton, S. and Devine, K. (2009). MapReduce-MPI Library Users Manual. Technical report, Sandia National Laboratories, USA.

Snir, M., Otto, S., Steven, H., Walker, D., and Dongarra, J. (1996). MPI: The complete reference. Technical report, MIT Press, Cambridge MA.

Tang, B., Moca, M., Chevalier, S., He, H., and Fedak, G. (2010). Towards MapReduce for Desktop Grid Computing. *IEEE Intern. Conf on P2P, Parallel, Grid, Cloud and Internet Computing*, pages 193–200.

Thung, F., Lo, D., and Jiang, L. (2012). Aumotatic Defect Categorization. *19th Working Conference on Reverse Engineering*, pages 205–214.

Wang, Y. and Wei, J. (2011). VIAF:Verification-based Integrity Assurance Framework for MapReduce. *IEEE Intern. Conf. on Cloud Computing. Miami, USA*, pages 300–307.

White, T. (2012). *Hadoop: The Definitive Guide*, volume 1. Yahoo Press, www.it-ebooks.info, 3 ed. edition.

Zheng, J., Nagappan, N., Hudepohl, J. P., and Vouk, M. A. (2006). On the Value of Static Analysis for Fault Detection in Software. *IEEE Trans. on Soft. Engin.*, 32(4):240–253.