4.
O mecanismo de comparação híbrido

Este capítulo apresentará o mecanismo de comparação produzido durante esta pesquisa. Inicialmente introduziremos conceitos gerais e nomenclaturas, de cunho geral e definidos por esta dissertação, relacionados à comparação de documentos. Em seguida será descrito o formato da estrutura sintática que o algoritmo de comparação espera receber e a estrutura utilizada para retornar as diferenças identificadas. Com isso podemos descrever na seção seguinte o algoritmo criado para comparar tais estruturas. Finalmente, o capítulo será concluído com uma comparação do mecanismo apresentado com o estado atual da arte previamente discutido.

4.1
Conceitos gerais e nomenclaturas

Em geral, a comparação de duas versões de um documento resulta em um conjunto de diferenças que expressam o que foi modificado entre as versões comparadas. Cada uma destas diferenças deve no mínimo identificar os limites do conteúdo modificado, expressando em uma *seleção* os limites em um sistema de coordenadas. Vamos assumir que na comparação de documentos textuais uma coordenada é definida por um *ponto*, que é composto por atributos *linha* e *coluna*. Então, uma *seleção* é composta por um ponto inicial e um ponto final.

Também é comum identificar o tipo de operação que cada diferença representa, e isso é feito mesmo em ferramentas mais primitivas. A partir do estudo de trabalhos na literatura e da análise de ferramentas comerciais, temos que os tipos de operação mais comuns que podem ser detectados são:

- Inserção – Um trecho de código existe na versão nova, sem correspondente na versão antiga.
- Remoção – Um trecho de código existe na versão antiga, sem correspondente na versão nova.
- Modificação – Um trecho de código na versão antiga possui um correspondente na versão nova, porém com algumas diferenças.

- Movimentação – Um trecho de código na versão antiga possui um correspondente na versão nova, porém em uma posição diferente do documento.
- União – Dois trechos de código na versão antiga possuem um único correspondente na versão nova, juntando neste as porções de código da versão antiga.
- Separação – Um trecho de código na versão antiga possui dois correspondentes na versão nova, sendo que cada um representa uma parte do trecho antigo.
- Clone – Um trecho de código em uma das versões possui mais de um correspondente na outra versão, e a união destes não representa totalmente o trecho unitário em uma das versões.

As ferramentas de comparação tradicionais costumam identificar no mínimo os tipos *Inserção* e *Remoção*. Algumas ferramentas textuais um pouco mais elaboradas são capazes de identificar operações de *Modificação*. Porém, ainda são resultados inadequados, devido à falta de precisão nos limites da diferença e falta de respeito pela hierarquia e pelo tipo de objeto comparado. Ferramentas tradicionais identificam as diferenças a partir da *menor distância de edição textual*, e o mecanismo proposto procura identificar a *menor distância de edição sintática*, ou seja, encontrar o menor número de diferenças respeitando os limites de cada elemento sintático.

Este trabalho busca identificar com precisão as operações *Inserção*, *Remoção*, *Modificação* e *Movimentação*. Além disso, procura prover mecanismos para, futuramente, implementar a detecção das operações de *União*, *Separação* e *Clone*, com base em heurísticas.

Uma visão geral do processo interno da ferramenta é expressado pelo diagrama da Figura 5.

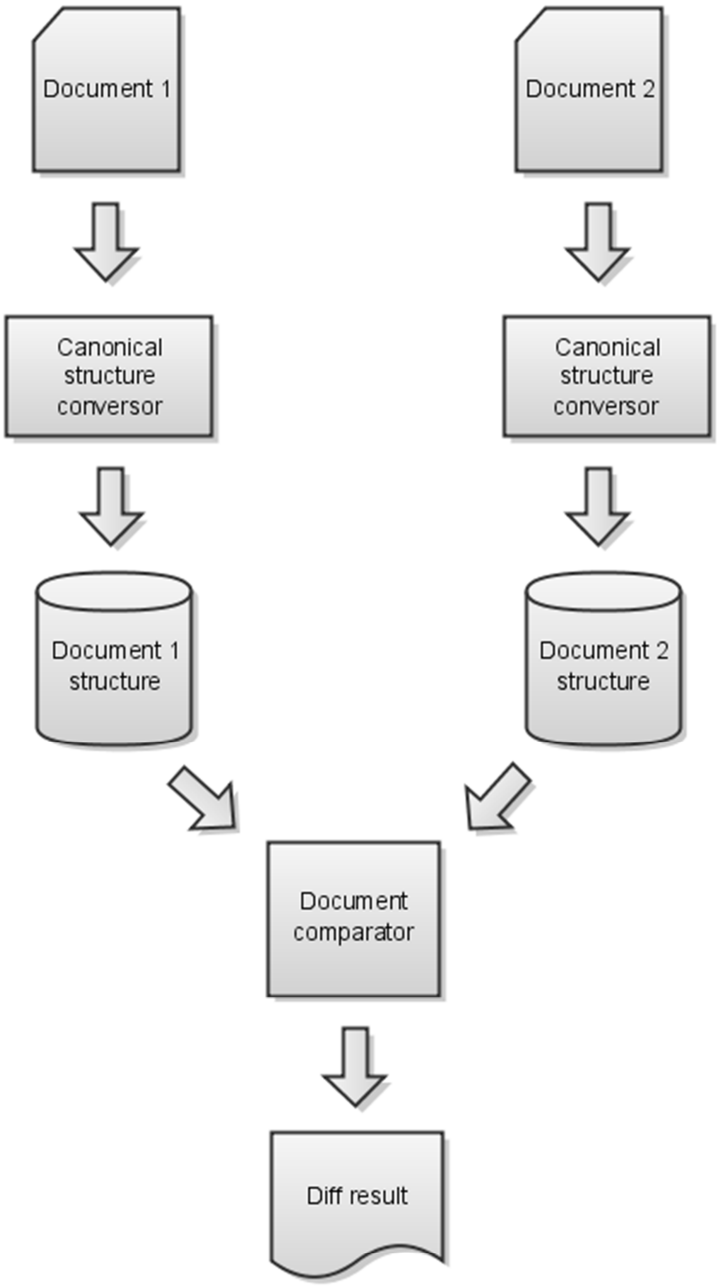

Figura 5 – Processo realizado pelo mecanismo de comparação

De forma sucinta, o modelo proposto transforma o conteúdo dos documentos a serem comparados em uma forma canônica, e em seguida o algoritmo de comparação genérico, pois trabalha sobre esta forma canônica, realiza a comparação das estruturas e retorna uma estrutura *Diff* (descrita nas seções seguintes) contendo informações sobre as diferenças encontradas.

4.2
Estrutura sintática canônica

O algoritmo de comparação trabalha sobre uma estrutura sintática canônica, a fim de mantê-lo genérico para qualquer tipo de documento textual[1]. A estrutura deve representar os elementos sintáticos presentes no documento e seus relacionamentos hierárquicos. Um exemplo de uma estrutura representando um trecho de código de programação é apresentado na Figura 6.

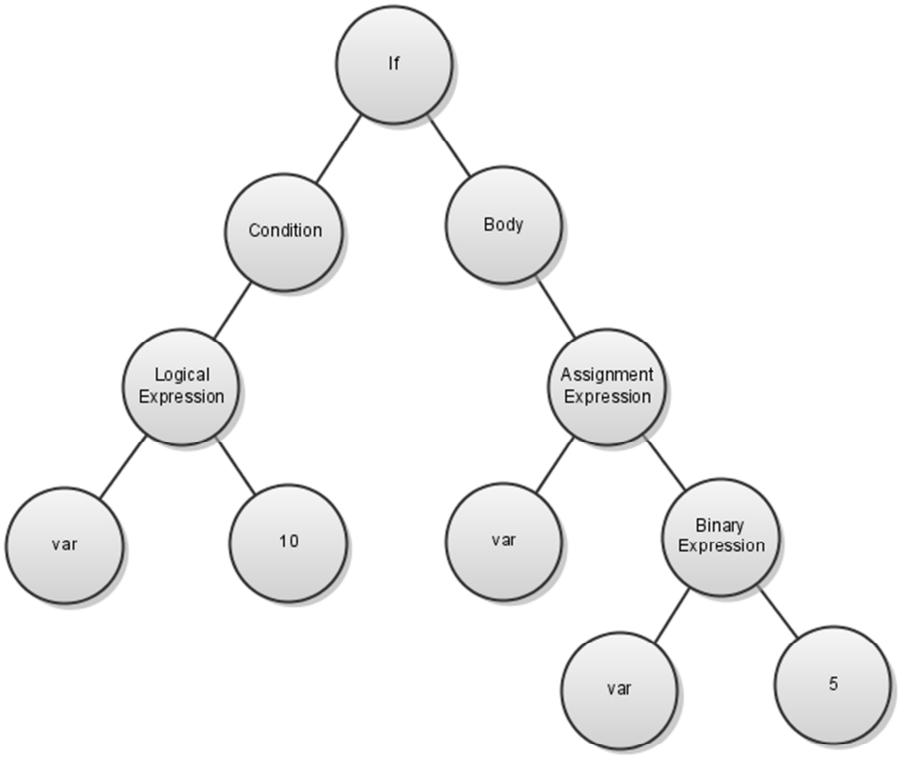

Figura 6 – Exemplo da estrutura de um trecho de código em uma linguagem de programação qualquer.

[1] A restrição a documentos textuais se deve unicamente ao tipo de resposta utilizado que armazena, em coordenadas textuais, os limites do texto que sofreu modificação, a fim de concentrar em uma única estrutura todas as informações necessárias para exibir o resultado em uma interface gráfica. Porém, com pouco esforço é possível generalizar o algoritmo com uma hierarquia de classes onde as características textuais fiquem presentes apenas em um nível mais específico.

Esta estrutura canônica é gerada por um conversor que recebe o conteúdo específico, interpreta-o, e retorna sua representação estrutural, respeitando e adicionando propriedades sintáticas identificadas durante o processo. Também são adicionadas algumas propriedades textuais, como veremos em breve. O resultado do conversor é retornado em XML [QUIN, 2003], um formato flexível e genérico suficiente para representar a grande maioria dos documentos. Esta estrutura em XML retornada é a estrutura canônica, que será entregue ao algoritmo de comparação e este a carregará em memória em uma composição de objetos, a estrutura utilizada na comparação.

Como exemplo, assuma que o código abaixo representa a declaração de um atributo em uma classe:

```
30.   private:
31.       int m_var;
```

Figura 7 – Trecho de código utilizado como exemplo.

Ao converter essa declaração o XML resultante será:

```
<VariableDeclaration
      visibility="private"
      startLine="31"
      startColumn="5"
      endLine="31"
      endColumn="15" >

      <TypeSpecifier
            value="int"
            startLine="31"
            startColumn="5"
            endLine="31"
            endColumn="8" />

      <Identifier
            value="m_var"
            startLine="31"
            startColumn="9"
            endLine="31"
            endColumn="14" />

</VariableDeclaration>
```

Figura 8 – XML gerado a partir do trecho de código apresentado na Figura 7.

Como pode ser verificado, a estrutura sintática do trecho de código apresentado na Figura 7 foi refletida no conteúdo XML apresentado na Figura 8, que é constituído por: um nó definindo uma declaração de variável, cujos filhos são um tipo e um identificador, que representam respectivamente o tipo e o nome

da variável que está sendo declarada. Estes, por conseguinte, não possuem filhos (nós terminais), mas guardam em um atributo o valor que expressam. Também é representado neste texto XML a visibilidade da declaração da variável, como uma propriedade do nó inicial desta estrutura. E, também, são armazenadas propriedades textuais indicando a *seleção* de cada elemento sintático no documento.

Uma observação importante é que cada conversor criado deve gerar uma árvore contendo um estrutura completa, com todas as propriedades do documento, a fim de viabilizar a operação inversa: linearizar o conteúdo estruturado em texto novamente. Com isso podemos, futuramente, reaproveitar o conteúdo estruturado para criar ferramentas como formatadores de código diretamente associados a um novo modelo de controle de versão, que persista o conteúdo estruturado em vez de texto puro.

4.3 Mecanismo de comparação

Esta subseção irá descrever em detalhes o mecanismo de comparação proposto. Inicialmente será descrita a estrutura de dados utilizada para representar o conteúdo estruturado. Em seguida apresentaremos a estrutura que armazena as diferenças identificadas. E, finalmente, será apresentado em detalhes o algoritmo de comparação sintática.

4.3.1 Estruturas *Document* e *DocumentElement*

O algoritmo recebe como entrada dois conteúdos no formato XML gerados pelo conversor. Cada um destes conteúdos XML é convertido para uma estrutura de dados interna (Figura 9), que armazena todas as informações necessárias para o algoritmo de comparação.

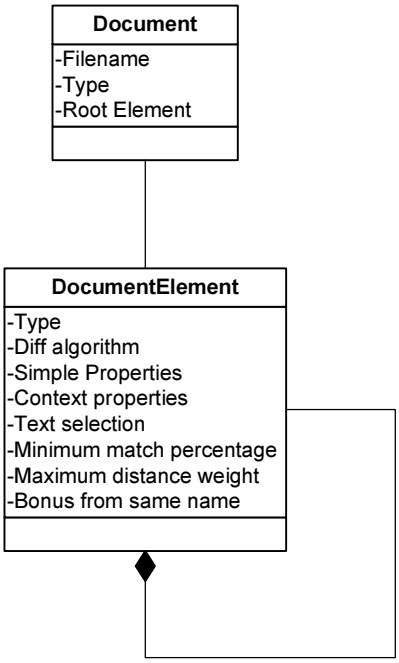

Figura 9 – Estrutura interna utilizada pelo algoritmo de comparação.

Esta estrutura de dados é composta por uma entidade *Document* representando a cabeça de uma árvore e tantas entidades *DocumentElement* quanto forem necessárias para representar cada nó da árvore. Cada entidade *DocumentElement* guarda o seu tipo sintático (declaração de variável, função, parâmetro, identificador, modificador, expressão lógica, etc.) e o algoritmo que deve ser usado para compará-lo. Como veremos nas seções seguintes, o algoritmo proposto é híbrido e permite configurar uma estratégia de comparação para cada tipo de elemento.

Cada atributo presente no conteúdo XML de entrada é classificado como uma propriedade simples ou uma propriedade de contexto. A diferença entre elas é que uma propriedade simples representa uma característica definida dentro do elemento sintático, e uma propriedade de contexto representa uma propriedade que atua sobre o elemento devido a ele estar em um determinado conjunto. A motivação para essa classificação será explicada na seção que descreve a comparação de propriedades.

A estrutura do elemento também mantém a *seleção* do texto que é representada por ele. Esta seleção é construída com os atributos *startLine*, *startColumn*, *endLine* e *endColumn* presentes em todos os nós do XML, previsto na forma canônica.

A utilização das propriedades "*Minimum match percentage*", "*Maximum distance weight*" e "*Bonus from same name*" serão explicadas na seção 4.3.3.3, em que serão descritas as heurísticas aplicadas.

4.3.2 Estrutura *Diff*

O algoritmo de comparação, a cada passo, compara um par de elementos e guarda as diferenças identificadas em uma estrutura *Diff*, apresentada na Figura 10. Como a comparação é executada recursivamente em uma árvore, para cada par de elementos é criada uma estrutura *Diff*. Estas estruturas são compostas ao longo da execução do algoritmo formando uma árvore de diferenças. Ou seja, até a resposta do algoritmo respeita as propriedades hierárquicas dos documentos. Logo, ao comparar os dois primeiros elementos de uma estrutura canônica, o resultado é uma composição de estruturas *Diff* contendo todas as diferenças identificadas entre os documentos.

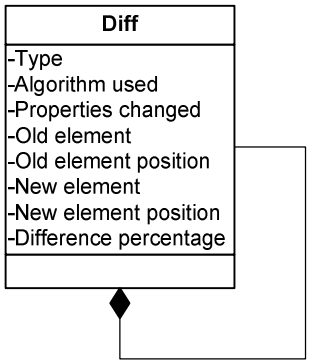

Figura 10 – Estrutura que representa a resposta do mecanismo de comparação.

Em relação as propriedades desta estrutura, a principal é o tipo de diferença identificada, que corresponde a operação que justifica a diferença. Esta propriedade pode assumir os valores:

- *Equal*
- *Inserted*
- *Removed*
- *Moved*
- *Modified*
- *Impossible comparison*

A última opção, *Impossible comparison*, é um caso especial, utilizado internamente pelo algoritmo de comparação para notificar quando dois elementos não podem ser comparados, devido aos seus tipos serem diferentes.

Esta estrutura de resposta armazena também uma referência para cada elemento comparado e qual estratégica de comparação foi utilizada para gerá-la. E, além das referências para cada elemento, também é registrado seu índice na lista de sub-elementos.

Finalmente, também é guardado o percentual de diferença calculado para o par de elementos comparados. Esta é a propriedade mais relevante para a tomada de decisão do algoritmo e possivelmente para se obter um bom resultado final. Este valor é calculado com base em três heurísticas apresentadas na seção que descreve a comparação sintática de elementos.

Para exemplificar esta estrutura de resposta, assuma que as estruturas sintáticas da figura abaixo serão comparadas:

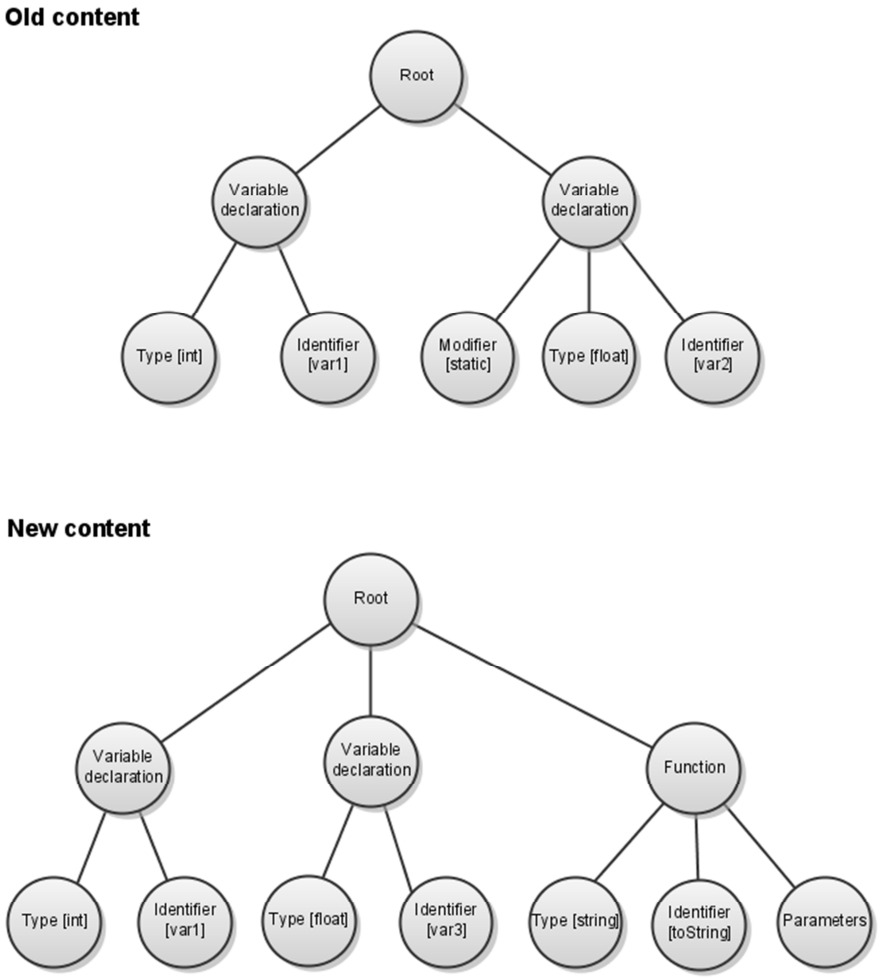

Figura 11 – Estruturas sintáticas que representam trechos de código a serem comparados para exemplificar a estrutura de resposta.

A estrutura resultante será:

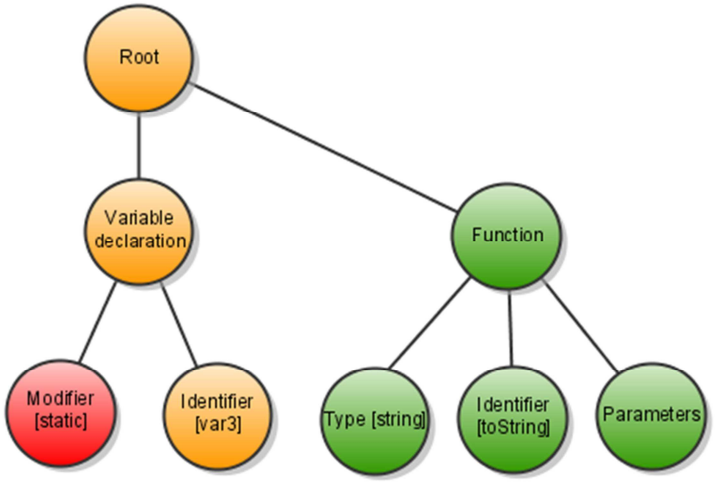

Figura 12 – Estrutura de diferenças resultante da comparação das estruturas apresentadas na Figura 11.

Como pode ser observado, o resultado é uma arvore identificando apenas os caminhos, das árvores de entrada, que apresentam alguma alteração. Para expressar o tipo de operação neste exemplo colorimos cada elemento segundo a tabela:

	Inserção de elementos
	Remoção de elementos
	Modificação de elementos

Neste exemplo, a estrutura sintática foi alterada pela remoção do modificador *static*, pela troca do nome da variável *var2* para *var3* e pela adição de uma declaração de função.

Concluindo, com essa estrutura de resposta preservamos as características hierárquicas dos documentos comparados por expressar o resultado também em uma estrutura hierárquica.

4.3.3 Algoritmo de comparação

Esta seção apresentará em detalhes o algoritmo de comparação de estruturas desenvolvido durante este trabalho. O algoritmo recebe como dados de entrada duas estruturas *Document* e retorna uma estrutura *Diff* contendo as diferenças identificadas, ambas as estruturas apresentadas nas seções anteriores.

O algoritmo de *comparação de elementos* funciona da seguinte forma: cada par de elementos é comparado por um procedimento, que, internamente chama um segundo procedimento para comparar seus sub-elementos, que por sua vez, é definido de acordo com o tipo de elemento que está sendo comparado. Este segundo procedimento que compara os sub-elementos é livre pra invocar recursivamente o método de *comparação de elementos* a fim de tratar de cada sub-elemento individualmente. Neste trabalho definimos dois tipos de comparação de sub-elementos:

1. Textual - aplica a comparação tradicional, com algumas melhorias realizadas em um pós-processamento.
2. Sintática – compara os sub-elementos considerando todas as suas propriedades sintáticas.

Desta forma, o algoritmo de *comparação de documentos* se inicia com a comparação dos elementos da cabeça de cada árvore, seguindo recursivamente nos elementos intermediários da arvore, até: (1) encontra um nó terminal (sem sub-elementos) ou (2) o segundo procedimento, de comparação de sub-elementos, não invocar a chamada recursiva.

O *pseudo-código* do algoritmo de *comparação de elementos* é exibido a seguir a fim de eliminar quaisquer dúvidas sobre o seu funcionamento, e introduzir suas etapas:

1. Comparação das propriedades dos elementos.
2. Comparação dos sub-elementos segundo a estratégia escolhida pelo tipo do elemento.
3. Calculo do percentual modificado.
4. Definição do tipo de resposta.

Cada uma destas etapas será abordada a seguir ou nas seções seguintes.

```
1   Diff compare_elements(
2           DocumentElement oldElement,
3           DocumentElement newElement)
4   {
5
6           Diff result;
7
8           if(oldElement.type == newElement.type) {
9
10                  // Compara as propriedades dos elementos
11                  compareProperties(oldElement, newElement, result);
12
13                  // Compara os sub-elementos segundo o algoritmo indicado
14                  // pelo tipo do elemento
15                  CompareChildAlgorithm algorithm =
16                                          getAlgorithmFrom(oldElement.type());
17                  algorithm.compareChilds(oldElement, newElement, result);
18
19                  // Calcula e guarda o percentual modificado
20                  double totalChanges = numPropertiesChanged + numDiffs;
21                  double totalPossibleChanges =
22                          totalPossiblePropertiesChanged + totalPossibleDiffs;
23                  result.setDifferencePercentage(
24                          totalChanges / totalPossibleChanges);
25
26                  // Guarda o algoritmo utilizado para comparar os
27                  //sub-elementos
28                  result.setAlgorithmUsed(algorithm.name);
29
31                  // Guarda o tipo da diferença
32                  DiffType type = DIFF_TYPE_EQUAL;
33                  if (differencePercentage != 0) {
34                          type = DIFF_TYPE_MODIFIED;
35                  }
36
37                  result.setType(type);
38
39          } else {
40
41                  // Se os tipos dos elementos são diferentes,
42                  // automaticamente eles são 100% diferentes
43                  result.setDifferencePercentage(100);
44                  result.setType(DIFF_TYPE_IMPOSSIBLE_COMPARISON);
45
46          }
47
48          return result;
49  }
50
```

Figura 13 – *Pseudo-código* do algoritmo de comparação de elementos.

Analisando este *pseudo-código*, a comparação se inicia com a comparação das propriedades, seguida da comparação dos sub-elementos, cujo algoritmo utilizado é definido a partir do tipo do elemento que está sendo comparado. Após estas duas etapas de comparação, para cada uma individualmente, é computado o número de diferenças e o número total de possíveis diferenças. Este número de diferenças é a *distância de Damerau* [DAMERAU, 1964], baseada na *distância de Levenshtein* [LEVENSHTEIN, 1966], que computa o menor número de operações para uma string tornar-se idêntica a outra. Levenshtein considera apenas operações de inserção e remoção de caracteres, enquanto Damerau

trabalha também com operações de substituição e transposição de caracteres. Ambos os algoritmos foram criados para tratar strings, porém podem ser modificados para atuar sobre sequências de elementos genéricos, como é o caso deste trabalho.

Com estas diferenças computadas, podemos calcular o percentual modificado usando a formula:

$$Percentual\ modificado = \frac{propriedades\ modificadas\ + subElementos\ modificados}{propriedades\ total + subElementos\ total}$$

Este valor computado indica o percentual de diferença entre as estruturas. A unidade de diferença para cada etapa de comparação pode ser variável: na comparação de propriedades cada propriedade representa uma unidade, porém na comparação de sub-elementos a unidade varia de acordo com o tipo de algoritmo. Como exemplo, temos que o algoritmo de comparação textual de sub-elementos usa o caractere como unidade de diferença.

Adiantando, este percentual modificado calculado para cada comparação é utilizado pelo algoritmo de comparação sintática de sub-elementos para decidir se um par de diferenças [*INSERT*, *REMOVE*] é na realidade uma única diferença do tipo *MODIFIED*. Ou seja, este percentual exerce um papel fundamental na melhoria de qualidade do resultado final, por viabilizar a identificação de trechos movidos ou modificados.

Observe que neste pseudo-código os tipos possíveis são *EQUAL*, *MODIFIED* e *IMPOSSIBLE_COMPARISON*. Porém, os tipos *INSERT* e *REMOVE* não são mencionados, pois são identificados dentro dos algoritmos de comparação de sub-elementos, como veremos em seções seguintes.

4.3.3.1 Comparação de propriedades

A primeira etapa de comparação é a comparação das propriedades de cada elemento. Já foi mencionado que temos dois tipos de propriedades:

1. Simples - representam características definidas dentro dos limites sintáticos do elemento.
2. Contextual - representam características que atuam sobre o elemento devido a ele estar imerso em um determinado conjunto.

Como exemplo, vamos classificar as propriedades do elemento XML da Figura 15:

```
24. public:
25.   MyClass();
26.   ~MyClass();
27.
```

Figura 14 – Exemplo de código utilizado para exemplificar a classificação de propriedades.

```
<Destructor
      visibility="public"
      startLine="26"
      startColumn="5"
      endLine="26"
      endColumn="32" >

      <Identifier
            value="MyClass"
            startLine="26"
            startColumn="6"
            endLine="26"
            endColumn="29" />
      <Parameters
            startLine="26"
            startColumn="29"
            endLine="26"
            endColumn="31" />
</Destructor>
```

Figura 15 – Exemplo do XML que representa o elemento sintático da declaração do destrutor na linha 26 da Figura 14.

O primeiro atributo encontrado é *visibility*, com valor *public*. Este atributo é classificado como uma propriedade contextual, pois não pode ser encontrado na representação textual do elemento. Verifique na Figura 14 que a definição de visibilidade não está dentro da caixa vermelha, que representa os limites da estrutura sintática do destrutor. Isso porque na linguagem de programação C++ a declaração de visibilidade pode ser definida uma só vez e aplicada a todas as declarações seguintes, até uma nova declaração de visibilidade redefinir a corrente. Ou seja, cada declaração no corpo de uma classe guarda uma

propriedade que indica sua visibilidade, definida pelo contexto no qual está inserida.

O segundo atributo encontrado, *value*, representando o nome do elemento *Identifier* do destrutor, é classificado como uma propriedade simples, pois sua representação textual pode ser encontrada dentro dos limites sintáticos do elemento, a string “MyClass”.

Porém, o motivo ainda não explicado para classificar as propriedades nesses dois tipos é permitir que estas possam ser selecionadas para comparação de acordo com a estratégia de comparação corrente. A motivação para isso é evitar que propriedades simples sejam comparadas duas vezes quando utilizado um mecanismo de comparação textual. Explicando, esta situação ocorre quando um fragmento sintático presente no texto do elemento é comparado como: (1) uma propriedade, e, posteriormente, (2) como uma informação textual presente nos limites sintáticos do elemento. Logo, em uma comparação textual selecionamos apenas as propriedades de contexto, ou seja, as que não apresentam fragmentos textuais dentro dos limites sintáticos dos elementos. Como exemplo, observe que se o elemento *Identifier* do exemplo for comparado utilizando uma estratégia de comparação textual o nome do destrutor será comparado duas vezes: na comparação de propriedades e na comparação do texto do elemento. Desta forma, temos que a comparação de propriedades escolhe quais serão comparadas de acordo a estratégia de comparação do elemento.

4.3.3.2 Comparação textual

Como mencionado, a partir do tipo de cada elemento é definido qual algoritmo de comparação será usado para comparar seus sub-elementos. Esta seção apresentará o mecanismo tradicional de comparação textual, com acréscimo de um pós-processamento a fim de melhorar a qualidade do resultado. Este mecanismo será apresentado antes do de comparação sintática como uma introdução, devido a sua simplicidade.

O algoritmo de comparação textual pode ser expresso pelo *pseudo-código* apresentado na Figura 16:

```
1   void compare(
2           DocumentElement oldElement,
3           DocumentElement newElement,
4           Diff result)
5   {
6           // Pega os trechos de texto
7           String oldText = oldElement.text();
8           String newText = newElement.text();
9
10          // Executa o algoritmo LCS tradicional sobre os trechos de texto
11          LCSResult lcsResult = lcs(oldText, newText);
12
13          // Avalia o resultado do algoritmo LCS em busca de inserções,
14          // remoções e modificações
15          TextDiffList textDiffList = evaluateDiffs(lcsResult);
16
17          // Elimina diferenças vazias, compostas por espaços, tabulações,
18          // quebras de linha, etc...
19          removeEmptyDiffs(textDiffList);
20
21          // Unifica diferenças muito próximas
22          mergeCloseDiffs(textDiffList);
23
24          // Preenche o resultado criando uma estrutura Diff para cada
25          // diferença textual encontrada.
26          fillResult(textDiffList, result);
27
28          // Calcula o número de caracteres e o número total de possíveis
29          // modificações
31          calculateNumCharatersAndTotalPossibleChanges(
32                  textDiffList, oldText, newText);
33  }
34
```

Figura 16 – *Pseudo-código* do algoritmo de comparação textual.

Inicialmente o algoritmo LCS [CORMEN, *et al.*, 2001] é executado sobre a representação textual dos elementos. O resultado deste algoritmo (*lcsResult*) é uma matriz com marcadores (*esquerda*, *cima* e *diagonal*) que contém, além da solução de subsequência máxima, as operações de inserção e deleção de caracteres que devem ser realizadas para que uma string se transforme na outra, e, são essas operações que temos interesse. Para cada conjunto de operações consecutivas criamos uma estrutura *TextDiff* (Figura 17). Desta forma, o resultado da comparação realizada pelo algoritmo LCS é uma lista de *diferenças textuais* composta por estruturas *TextDiff*.

```
1   TextDiff
2   {
3           // Tipo da operação.
4           TextDiffType m_operation;
5           // Índice do caractere inicial do texto na string antiga.
6           int m_oldStart;
7           // Índice do caractere final do texto na string antiga.
8           int m_oldEnd;
9           // Índice do caractere inicial do texto na string nova.
10          int m_newStart;
11          // Índice do caractere final do texto na string nova.
12          int m_newEnd;
13  };
14
```

Figura 17 – Definição da estrutura *TextDiff*.

Cada estrutura *TextDiff* guarda o tipo de operação e os limites das diferenças em cada string. Os tipos possíveis são *Inserção*, *Remoção* e *Modificação*, sendo que este último é inferido a partir de um padrão identificado, explicado a seguir. Em relação aos limites, estes são definidos pelas posições inicial e final da diferença em cada string. Perceba que em diferenças cuja operação seja uma *Inserção* ou *Remoção*, apenas uma string é modificada, deixando a outra com uma seleção vazia.

Para exemplificar, suponha que vamos comparar as strings “myVariable” e “myConstVariable”: o algoritmo LCS informará que os caracteres [C, o, n, s, t] devem ser inseridos nas posições [2, 3, 4, 5, 6], respectivamente, para transformar a primeira string na segunda. Estas totalizam 5 operações de inserção de caracteres, que serão agrupadas em uma única estrutura *TextDiff*, marcada com o tipo de operação *Inserção*, e com limites [2, 2] na primeira string e [2, 7] na segunda string.

Como já foi mencionado, o algoritmo identifica operações do tipo *Modificação* a partir de um padrão: sempre que encontrarmos uma diferença do tipo *Inserção* seguida de uma diferença *Remoção*, se elas forem consecutivas, ou seja, se não existir nenhum caractere entre elas, podemos fundi-las em uma operação de *Modificação*. Vamos tomar como exemplo a comparação das strings “myConstVariable” e “myStaticVariable”: o algoritmo LCS irá indicar a remoção dos caracteres [C, o, n, s, t] na primeira string, com limites [2, 7]; e a inserção dos caracteres [S, t, a, t, i, c] na segunda string, com limites [2, 8]. Inicialmente estas diferenças são identificadas como uma remoção e uma inserção, porém como são consecutivas no resultado do algoritmo LCS, unificamos em uma única diferença que representa uma apenas uma modificação.

O primeiro pós-processamento realizado é buscar por diferenças textuais vazias, ou seja, que sejam compostas por espaços, tabulações e quebras de linha. Essas diferenças apenas dificultam a leitura e o entendimento das diferenças, logo, excluímo-las da resposta.

O segundo pós-processamento é a identificação de diferenças muito próximas, que podem ser fundidas em uma única. Para isso percorremos a lista de diferenças encontradas e para cada uma verificamos, para cada trecho de texto separadamente, se a distância entre o final dele e o início do próximo é inferior a um valor calculado com base no tamanho total se os trechos de texto forem

unificados. Para auxiliar a compreensão deste procedimento, expressamos ele no *pseudo-código* da Figura 18:

```
1
2   const double MAXIMUM_PERCENTAGE_MULTIPLYER = 0.1;
3
4   for (int i = 0; i < textDiffList.size() - 1; i++) {
5           TextDiff diff = textDiffList[i];
6           TextDiff nextDiff = textDiffList[i + 1];
7
8           // Pego o tamanho total se as strings forem unificadas
9           int oldUnifiedSize = nextDiff.oldEnd() - diff.oldStart();
10          int newUnifiedSize = nextDiff.newEnd() - diff.newStart();
11
12          // Pego o tamanho dos intervalos entre as strings
13          int oldIntervalSize = nextDiff.oldStart() - diff.oldEnd();
14          int newIntervalSize = nextDiff.newStart() - diff.newEnd();
15
16          // Calculo o tamanho máximo aceitável para cada intervalo
17          int oldMaximumIntervalSize =
18                  oldUnifiedSize * MAXIMUM_PERCENTAGE_MULTIPLYER;
19          int newMaximumIntervalSize =
20                  newUnifiedSize * MAXIMUM_PERCENTAGE_MULTIPLYER;
21
22          // Se ao menos um dos intervalos for menor que o seu respectivo
23          // intervalo máximo, unificamos as diferenças.
24          if ((oldIntervalSize <= oldMaximumIntervalSize)
25          ||  (newIntervalSize <= newMaximumIntervalSize)) {
26
27                  // Redefino os limites finais da primeira diferença
28                  diff.setOldEnd(nextDiff.oldEnd());
29                  diff.setNewEnd(nextDiff.newEnd());
31
32                  // A diferença passa a ser uma modificação caso
33                  // os tipos sejam diferentes
34                  if (diff.operation() != nextDiff.operation()) {
35                          diff.setOperation(MODIFY_TEXT);
36                  }
37
38                  // Removo a segunda diferença da lista
39                  textDiffList.removeAll(nextDiff);
40          }
41  }
42
```

Figura 18 – *Pseudo-código* do pós-processamento de união de diferenças próximas.

Finalmente, cada estrutura representando uma *diferença textual* é transformada em uma estrutura *Diff*, apresentada anteriormente. Para auxiliar a explicação desta etapa, apresentamos o seu *pseudo-código* na Figura 19. Resumidamente, os limites encontrados em cada diferença textual são convertidos em duas *seleções*, que indicam os trechos de código modificados em cada elemento comparado. Estas seleções são ajustadas, removendo espaços em branco em torno delas. Em seguida, cada seleção é utilizada para construir um elemento *dummy*, a fim de representar um elemento sintático que originou a diferença. Isso é necessário para manter conformidade com o algoritmo de comparação sintática em que os elementos sintáticos comparados são registrados na estrutura *Diff*. O procedimento termina com a criação de uma estrutura *Diff* e o registro nela dos

elementos sintáticos previamente criados. Neste momento também é registrado seu tipo e é calculado o percentual de diferença. Cada estrutura *Diff* criada por este procedimento é registrada como filha da estrutura *Diff* recebida como entrada do algoritmo.

```
1   // Para cada diferença encontrada criamos um elemento texto contendo
2   // a seleção
3   for (int i = 0; i < textDiffList.size(); i++) {
4           TextDiff textDiff = textDiffList[i];
5
6           int start1 = textDiff.oldStart();
7           int end1 = textDiff.oldEnd();
8           int start2 = textDiff.newStart();
9           int end2 = textDiff.newEnd();
10
11          // Define qual operação foi realizada
12          DiffType type;
13          TextDiffType operation = textDiff.operation();
14          if(operation == INSERT_TEXT) {
15                  type = DIFF_TYPE_INSERTED;
16          } else if(operation == REMOVE_TEXT) {
17                  type = DIFF_TYPE_REMOVED;
18          } else if(operation == MODIFY_TEXT) {
19                  type = DIFF_TYPE_MODIFIED;
20          }
21
22          // Converte as posições relativas em seleções
23          TextSelection oldSelectionResult =
24                  relativePositionToSelection(start1, end1);
25          TextSelection newSelectionResult =
26                  relativePositionToSelection(start2, end2);
27
28          // Remove os espaços em volta
29          trimSelectionLimits(oldSelectionResult);
31          trimSelectionLimits(newSelectionResult);
32
33          // Crio os elementos falsos para representar o texto modificado
34          FakeElement oldFakeElement(oldSelectionResult);
35          FakeElement newFakeElement(newSelectionResult);
36
37          // Crio o diff resultante
38          Diff diff = createDiff(oldFakeElement, newFakeElement, type);
39
40          // Registra o diff
41          result.append(diff);
42  }
43
```

Figura 19 – *Pseudo-código* do procedimento que transforma a lista de diferenças textuais em diferenças estruturais.

O algoritmo de comparação textual termina computando o número de caracteres modificados e o número total de possíveis modificações (soma dos tamanhos dos trechos de código comparados), a fim de calcular o percentual de diferença entre as estruturas, como apresentado na seção anterior.

4.3.3.3 Comparação sintática

Esta seção apresentará o algoritmo de comparação sintática de sub-elementos. Sua abordagem inicial é semelhante a do algoritmo de comparação

textual, aplicando um algoritmo LCS as listas de sub-elementos, porém usando como elemento de comparação a estrutura sintática (*DocumentElement*). O resultado obtido, como na comparação textual, é uma sequência indicando quais unidades devem ser inseridas, e quais devem ser removidas em uma das listas para transformá-la na outra. A partir do resultado deste algoritmo conseguiríamos construir uma lista de diferenças (estruturas *Diff*) com os tipos *Inserted* e *Removed* apenas, pois é o máximo que se consegue extrair com o algoritmo LCS. Porém, somente com este passo conseguimos atingir grande parte do objetivo deste trabalho, que é realizar a comparação de conteúdos respeitando sua estrutura sintática. Este resultado não apenas respeita os limites sintáticos, como impede que estruturas de tipos diferentes sejam comparadas (explicação na seção 4.3.3).

Porém, é possível melhorar drasticamente o resultado da comparação se formos capazes de inferir, dentre as operações de inserção e remoção, quais, aos pares, representam na realidade operações de modificação. Atingir este objetivo foi um avanço significativo que fizemos neste trabalho e que será descrito ao longo desta seção. A seguir, na Figura 20, apresentamos o *pseudo-código* do algoritmo de comparação sintática de sub-elementos a fim de auxiliar a explicação.

```
1   void compare(
2           DocumentElement oldElement,
3           DocumentElement newElement,
4           Diff result)
5   {
6           // Pega as listas de subElementos
7           ElementList oldElementList = oldElement.subelements();
8           ElementList newElementList = newElement.subelements();
9
10          // Executa o algoritmo LCS sobre as listas
11          LCSResult lcsResult = lcs(oldElementList, newElementList);
12
13          // Avalia o resultado do algoritmo LCS em busca de inserções
14          // e remoções
15          ElementList insertedElements, removedElements =
16                  processResult(lcsResult);
17
18          // Constroi a matriz de entrada para o algoritmo Hungarian
19          // utilizando apenas os elementos inseridos e removidos
20          HungarianMatrix matrix =
21                  buildHungarianMatrix(insertedElements, removedElements);
22
23          // Executa o algoritmo Hungarian, de emparelhamento, sobre a
24          // matriz construída
25          MatchingResult result = hungarian(matrix);
26
27          // Avalia o resultado do algoritmo de emparelhamento e gera a
28          // lista de diferenças
29          DiffList diffList = evaluateMatchingResult(result);
31          result.append(diffList);
32
33          // Identifica as operações de modificação que correspondem a
34          // operações de movimentação.
35          diffList = evaluateMoves(diffList);
36
37          // Calcula o número de elementos e o número total de possíveis
38          // modificações
39          calculateNumElementsAndTotalPossibleChanges(
40                  diffList, oldElementList, newElementList);
41
42  }
43
```

Figura 20 – *Pseudo-código* do algoritmo de comparação sintática de sub-elementos.

Conforme descrito, o algoritmo começa executando um algoritmo LCS sobre as listas de sub-elementos dos elementos sintáticos a serem comparados. Neste passo é realizada uma chamada recursiva de comparação de elementos para cada par de sub-elementos. O resultado desta comparação indica para o algoritmo LCS se dois sub-elementos são idênticos, verificando se o percentual diferente é igual a zero. Este procedimento parece custoso a primeira vista, porém verifique que somente são comparados elementos do mesmo tipo, e, que estão no mesmo nível hierárquico, reduzindo drasticamente o número de comparações, quando comparado com a abordagem em que é realizada a comparação de todos os elementos de uma árvore com todos os elementos da outra, ignorando suas propriedades estruturais. No capítulo 6 serão apresentadas algumas estatísticas

sobre a execução do algoritmo que comprovam esta redução no número de comparações.

Em seguida, este resultado é processado e gera duas listas: a dos elementos que devem ser removidos e a dos elementos que devem ser inseridos, tomando como referencial a primeira lista de sub-elementos (*oldElementList*). Ou seja, aplicando estas inserções e remoções sobre a primeira lista de sub-elementos ela se transforma na segunda.

O próximo passo é identificar quais elementos inseridos e removidos, aos pares, são equivalentes, e podem ser fundidos em uma operação de modificação ou movimentação. Como o mecanismo apresentado por este trabalho optou por uma solução que preservasse a liberdade do usuário, permitindo que este utilize qualquer ferramenta para editar o documento, a comparação é baseada apenas nos seus estados inicial e final. Ou seja, não guardamos identificadores persistentes em cada elemento sintático ou mesmo uma lista de operações de edição realizadas para auxiliar este procedimento.

Logo, o máximo que podemos fazer para identificar estas operações é inferi-las a partir de heurísticas, descritas detalhadamente a seguir, e que por sua vez, devem ser configuráveis de acordo com o contexto ao qual estão sendo aplicadas. Como exemplo, considerando o contexto de linguagens de programação, uma operação de movimentação de elementos no corpo de um método possui características diferentes de uma operação de movimentação de elementos no corpo de uma classe. Todas as heurísticas e suas propriedades configuráveis serão explicadas a seguir, porém, antes, é necessário entender a contribuição que uma heurística tem no cálculo de equivalência entre dois elementos. A lógica utilizada para calibrar as heurísticas será descrita na seção seguinte.

O objetivo desta etapa não é encontrar o par com maior equivalência, e sim encontrar a melhor combinação de equivalência entre os elementos inseridos e removidos. Ou seja, buscamos o emparelhamento cuja soma de todos os pesos seja máxima. O *peso* dentro do contexto do nosso problema é um valor calculado com base nas heurísticas para expressar o grau de igualdade entre dois elementos.

Para descobrir o emparelhamento com peso máximo utilizamos o algoritmo Hungarian [Kuhn, 1955] cuja complexidade é $O(n^3)$, maior que a complexidade do algoritmo LCS utilizado inicialmente, cuja complexidade é O(mn), porém, o número de elementos submetidos ao algoritmo de emparelhamento tende a ser

muito pequeno, consistindo nos elementos identificados como inseridos e removidos pelo algoritmo LCS. É possível afirmar isso, pois usualmente a evolução de uma versão tende a modificar poucos elementos.

Continuando a explicação do *pseudo-código*, construímos uma matriz em que cada linha representa um elemento removido e cada coluna representa um elemento inserido. Em cada célula guardamos o peso inferido para o par de elementos da sua respectiva linha e coluna. Este peso é um valor inteiro que pode assumir valores entre 0 e 100, e é calculado, como já mencionado, a partir das heurísticas criadas. Inicialmente o peso começa com valor zero (Figura 21), e vai sendo construído aplicando as heurísticas sobre ele. Nos trechos de código a seguir assumiremos que as variáveis *i* e *j* corresponderão aos índices dos elementos removidos e inseridos, respectivamente. Também consideraremos que a variável *diffMatrix* é uma matriz que guarda as estruturas *Diff* obtidas como resposta durante a execução do algoritmo LCS.

```
1  int weight = 0;
2
```

Figura 21 – Inicialização do peso de um par de elementos.

A primeira heurística a ser calculada é a **heurística baseada na semelhança dos elementos**, que é o inverso do percentual de diferença obtido na comparação dos elementos. Este é um valor que varia de zero, quando os elementos são totalmente diferentes, até 100, quando os elementos são idênticos. Seu cálculo, trivial, pode ser expresso pelo código na Figura 22.

```
1  weight += 100 – diffMatrix[i][j].differencePercentage();
2
```

Figura 22 – *Pseudo-código* da heurística baseada na semelhança dos elementos.

A segunda heurística é a **heurística baseada na distância dos elementos**, que busca considerar no peso a distância que os objetos estão dispostos. Esta heurística provoca uma redução no grau de igualdade dos elementos proporcional a sua distância, seu peso é configurável de acordo com cada contexto. A Figura 23 mostra um *pseudo-código* que representa esta heurística.

```
1  double distance = abs(i - j);
2  double maxDistance = max(oldChildElements.size(), newChildElements.size());
3  double distanceWeight = oldElement.maximumDistanceWeight();
4  double distanceCost = (distance / maxDistance) * distanceWeight;
5  weight -= distanceCost;
6  weight = max(value, 0);
7
```

Figura 23 – *Pseudo-código* da heurística baseada na distância dos elementos.

Para exemplificar, considere o código da Figura 24, onde removemos uma expressão próxima do início de um corpo de método e adicionamos outra próxima do final, muito semelhante a primeira. Queremos que elas sejam identificadas como uma inserção e uma remoção, em vez de uma modificação, devido à invocação da função *printf* em ambas as expressões. Esta semelhança faria com que elas fossem consideradas equivalentes, porém, ao aplicar a heurística baseada na semelhança, e, conseqüentemente, reduzir o valor do peso com a heurística baseada na distância, este par, apesar de ser um candidato ao emparelhamento, não atinge o limiar mínimo (*maximumDistanceWeight*) para que seus elementos sejam considerados equivalentes.

```
1  int main()                          int main()
2  {                                   {
3          f0();                               f0();
4          printf("Hello world");              f1();
5          f1();                               f2();
6          f2();                               f3();
7          f3();                               printf("My name is Thiago");
8          f4();                               f4();
9  }                                   }
10
```

Figura 24 – Exemplo da heurística baseada na distância dos elementos no caso que os elementos devem ser considerados diferentes.

Ao submeter o exemplo acima ao mecanismo de comparação, este notifica uma inserção e uma remoção, desde que a propriedade *maximumDistanceWeight* do elemento sintático *FunctionCall* seja configurada com um valor relativamente alto. Assim como nas outras heurísticas o valor dessa propriedade deve variar entre 0 e 100. Supondo que o elemento *FunctionCall* seja configurado com a propriedade *maximumDistanceWeight* igual a 80 (ou seja, penalizando bastante a distância dos elementos), o custo da distância para os elementos do exemplo da Figura 24 seria exatamente 60, calculado a partir do *pseudo-código* da Figura 23.

Observe que nem sempre queremos penalizar a distância dos elementos pois existem escopos em que a reordenação de elementos não altera a semântica, como

exemplo, um corpo de classe. Apresentamos na Figura 25 um exemplo que explora essa característica, promovendo a movimentação de uma declaração do início para o final de um corpo de classe, resultando, como esperado, em equivalência. Neste caso temos que o elemento sintático *Function* teve sua propriedade *maximumDistanceWeight* configurada com o valor 0, para que o peso devido a distância dos elementos resulte sempre em zero.

```
1  class MyClass                          class MyClass
2  {                                      {
3          int f1() { printf("a"); }              int f2() { printf("b"); }
4          int f2() { printf("b"); }              int f3() { printf("c"); }
5          int f3() { printf("c"); }              int f4() { printf("d"); }
6          int f4() { printf("d"); }              int f5() { printf("e"); }
7          int f5() { printf("e"); }              int f1() { printf("a+"); }
8  };                                     };
9
```

Figura 25 - Exemplo da heurística baseada na distância dos elementos no caso que os elementos não devem ser considerados diferentes.

A terceira e última heurística é a **heurística baseada nos nomes dos elementos**, que induz o algoritmo de comparação a definir como equivalentes dois elementos que possuam o mesmo nome. Para isso, introduzimos o conceito de elementos nomeáveis, cujo nome é uma propriedade opcional e definida pela própria lógica do elemento. Como exemplo, considerando o contexto de linguagens de programação, um elemento do tipo *Function* pode definir que seu nome é o valor do seu sub-elemento do tipo *Identifier*. Esta heurística é aplicada garantindo um bônus, configurável por tipo de elemento, a pares cujos nomes dos elementos sejam idênticos. O *pseudo-código* da Figura 26 auxilia a compreensão desta heurística.

```
1  if (oldElement.name() == newElement.name()) {
2          value += oldElement.bonusFromSameName();
3          value = min(value, 100);
4  }
5
```

Figura 26 - *Pseudo-código* da heurística baseada no nome dos elementos.

A matriz construída com os pesos é submetida ao algoritmo Hungarian e seu resultado é uma lista de pares, cuja soma dos pesos é a maior possível. Desta forma, conseguimos identificar dentre os elementos inseridos e removidos, o melhor emparelhamento.

O próximo passo é avaliar os pares emparelhados e definir quais realmente podem ser considerados equivalentes. Explicando melhor, não podemos considerar equivalente um par cujo percentual de igualdade seja 10%, mesmo que este tenha sido elegido pelo emparelhamento. A nossa solução é definir um limiar mínimo para considerar a equivalência entre os elementos, porém, este pode variar de acordo com cada contexto, ou seja, de acordo com o tipo do elemento comparado. Solucionamos isso definindo o valor do limiar mínimo em cada tipo de elemento, com a propriedade *minimumMatchPercentage*, e, todos os emparelhamentos cujo o peso for superior a este limiar, terão seus respectivos elementos considerados equivalentes. Esta propriedade também é baseada em heurísticas e, como já dito, a lógica de definição dos valores será abordada na seção seguinte.

O procedimento de comparação termina mapeando cada operação identificada em uma estrutura *Diff*, que em seguida é registrada como filha na estrutura *Diff* recebida como parâmetro do algoritmo de comparação sintática.

Finalmente, para cada operação de modificação avalia-se se esta representa uma operação de movimentação. Para inferir se o elemento foi movido para outra posição, comparamos a posição relativa do elemento em ambos os documentos, segundo a fórmula:

$$oldRelativePosition = oldPosition - numRemoves + numInserts - movesBefore$$

$$newRelativePosition = newPosition - numInserts + numRemoves - movesBefore$$

Ou seja, para cada elemento subtraímos o número de elementos que foram alterados antes da sua posição. Logo, se o resultado em ambos os documentos for diferente, a diferença é uma movimentação.

O último passo do algoritmo é, além de computar o número máximo de modificações possíveis, calcular o número de elementos modificados, que pode ser expresso pela formula:

$$elementsModified = \left(\sum_{0}^{numDiffs} diff.differencePercentage \right) \Big/ 100$$

Explicando, para computar o número de elementos modificados bastaria somar o número de diferenças, ou seja, o número de inserções, remoções e modificações. Porém, é errado contar como uma unidade inteira todas as operações de modificação, pois estas podem ter um percentual de diferença muito pequeno. O objetivo deste calculo é identificar quantos elementos inteiros foram modificados, e, para isso, somamos o percentual de diferença de cada um e dividimos por 100. Observe que se existirem apenas inserções e remoções o resultado será sempre um valor inteiro, idêntico ao número de diferenças, porém, em uma situação onde existem duas operações de modificação cujo percentual modificado é de 50%, temos que o resultado final é apenas um elemento modificado, em vez de dois. Concluindo, este calculo computa o número de diferenças proporcional, ou seja, considera que uma estrutura *Diff* representando uma modificação não conta como uma unidade modificada, mas sim uma parte de uma unidade, que, em conjunto com outras partes, pode formar um valor inteiro proporcional ao que foi modificado.

4.4 Escolha entre comparação textual ou sintática

Esta seção abordará a escolha da estratégia de comparação para cada elemento sintático. As possíveis estratégias, recém descritas, são: textual e sintática. A seguir descreveremos como chegamos ao resultado final, em vez de apenas expô-lo.

A primeira abordagem foi comparar sintaticamente todos os elementos até o escopo do corpo de métodos, e, o restante, textualmente. O resultado foi um alto grau de precisão nas diferenças identificadas a partir da estratégia de comparação sintática, porém, resultados inadequados nos elementos comparados textualmente, conforme apresentado nas motivações para este trabalho.

Com base no resultado da primeira abordagem, o segundo experimento foi configurar todos os elementos para serem comparados sintaticamente. Porém, o resultado também foi impreciso, pois, considerou elementos com pequenas diferenças como totalmente modificados. Como exemplo, considere a comparação de uma string literal com outra que foi ligeiramente modificada: a comparação sintática, por considerar o elemento inteiro, apresentará que o par é equivalente e

foi modificado, porém, não indicará com precisão o que foi alterado, ou seja, quais caracteres da string foram trocados.

Em suma, comprovamos na prática que um algoritmo híbrido gera resultados melhores. O experimento final resultou em sucesso quando definimos que os tipos de elementos que permitissem alterações parciais fossem comparados textualmente. Estes tipos são: comentário (simples e multilinha), string literal, identificador (nome de variável, nome de método, nome de atributo, etc.) e nome de tipo (nome de classe, nome de struct, nome de typedef, etc.). O resultado desse ajuste pode ser verificado no capítulo 6, onde apresentamos alguns exemplos gerados com a ferramenta.

4.5 Calibração das propriedades

Esta seção descreverá como as propriedades, utilizadas para guiar as heurísticas, foram calibradas para alcançar o resultado apresentado no capítulo 6, em que são descritos os experimentos realizados com a ferramenta criada durante este trabalho. Recordando, existem três tipos de propriedades que devem ser definidas por cada tipo de elemento:

- *MaximumDistanceWeight*
- *BonusFromSameName*
- *MinimumMatchPercentage*

Cada uma das propriedades deve ser configurada com um valor entre zero e 100, e calibradas segundo as explicações a seguir.

A primeira propriedade, *MaximumDistanceWeight*, é utilizada para penalizar pares emparelhados com um valor proporcional a distância entre eles. O valor configurado nesta propriedade é o valor do peso no caso extremo, em que a distância dos elementos é a maior possível: um elemento no início e outro no final nas listas de sub-elementos. Considerando o contexto de linguagens de programação, temos que esta propriedade deve ser configurada com um valor alto quando a reordenação dos elementos modifica a semântica, como por exemplo, chamadas de função no corpo de métodos; e, deve ser configurada com um valor baixo, quando a reordenação torna-se irrelevante para o contexto, como por exemplo nas declarações de métodos e atributos no corpo de uma classe. Porém, a propriedade é definida por cada tipo de elemento em vez de ser definida pelo

contexto no qual este elemento está inserido, pois é possível que em um mesmo contexto exista a necessidade de calibrar cada tipo de elemento com um valor diferente.

A segunda propriedade, *BonusFromSameName*, ao contrário da última apresentada, garante um bônus a elementos que possuam o mesmo nome, de acordo com as regras sintáticas do documento. A partir dos nosso experimentos encontramos que o valor desta propriedade deve ser configurado entre o valor médio e o valor máximo, ou seja, entre 50 e 100, para se obter bons resultados. Como exemplo, o elemento que representa uma chamada de função deve ser configurado com um valor baixo (próximo de 50), de forma que promova um bônus à chamadas da mesma função (nomes idênticos), porém, permita que o peso calculado a partir da distância e o percentual de diferenças atuem de forma que exista a chance dos elementos serem consideradas diferentes (se estiverem muito distantes ou forem muito diferentes). E, em um segundo exemplo, vamos considerar o elemento que representa a declaração de um método no corpo de uma classe: nele a propriedade deve ser configurada com um valor alto (próximo de 100), de forma que sempre seja emparelhado, pois, nesse contexto, por maior que seja a diferença entre os elementos se eles tiverem o mesmo nome provavelmente serão a mesma entidade.

Finalmente, a propriedade *MinimumMatchPercentage* é utilizada para definir se os elementos de um emparelhamento são equivalentes ou não: se o percentual de igualdade dos elementos for maior ou igual ao valor desta propriedade, estes são considerados o mesmo. Em geral, esta propriedade foi configurada com um valor alto (próximo de 70). As únicas exceções para esta valoração são os elementos únicos em seu contexto, como por exemplo, o elemento que engloba os parâmetros da declaração de uma função (todo elemento *Function* tem um único elemento *Parameters*). Os elementos que são únicos devem ter sua propriedade *MinimumMatchPercentage* configurada como zero, para que sejam ignorados na comparação, e, serem no máximo considerados modificados, quando seus sub-elementos apresentarem diferenças.

4.6
Comparação com o estado da arte

Comparando o modelo proposto por este trabalho com o estado da arte atual, temos que:

1. Optamos por um modelo não invasivo, evitando que o usuário final fique restrito a ferramentas especiais para editar o conteúdo dos arquivos, como nos trabalhos [PIETROBON, 1995] [LIPPE e OOSTEROM, 1992] [SHEN e SUN, 2001] [DIG, *et al.*, 2006] [DIG, NGUYEN e JOHNSON, 2006] [DIG, *et al.*, 2006]. Da forma escolhida ele fica restrito à ferramenta de comparação apenas na interação com o controle de versão, deixando seu ambiente de desenvolvimento livre para utilizar quaisquer ferramentas de edição.
2. O modelo trabalha sobre a representação sintática dos documentos, divergindo dos trabalhos que utilizam a AST [KLIER, 2005] [DIG, *et al.*, 2006] [DIG, *et al.*, 2005] [DIG, JOHNSON e MARINOV, 2006], com o objetivo de permitir que documentos incompletos (sem todas as referências resolvidas) possam ser interpretados e comparados. Exemplo: no contexto da linguagem C/C++ se uma determinada macro estiver definida, ela pode modificar o significado de uma expressão, porém, sintaticamente esta expressão é escrita da mesma forma.
3. O mecanismo de comparação respeita a hierarquia e os tipos dos elementos: somente são comparados elementos do mesmo tipo e que estejam em um nível da árvore equivalente, assim como em [NGUYEN, *et al.*, 2004] [JUNQUEIRA, BITTAR e FORTES, 2008] [WESTFECHTEL, *et al.*, 2001] [YANG, 1991] [KLIER, 2005] [MUNSON e DEWAN, 1994], evitando a comparação desnecessária de todos os elementos de uma árvore com todos os elementos da outra árvore.
4. A utilização de heurísticas combinadas ao algoritmo de emparelhamento é uma inovação proposta por este trabalho que evita resultados rígidos e direciona o resultado da comparação de cada

nível para possíveis operações de *Modificação* em vez de pares [*Inserção, Remoção*] equivalentes.

5. A proposta de Dig [DIG, *et al.*, 2006] relacionada à detecção de operações de refatoramento com base em padrões conhecidos é complementar ao nosso modelo e parte da proposta pode ser implementada na nossa ferramenta. Somente parte, pois a detecção de padrões que exijam uma AST completa não poderá ser realizada, pois como já mencionado, inspecionamos apenas o necessário para conhecer a estrutura sintática do documento. Porém, estimamos que a ausência destas informações possa ser solucionada com a criação de heurísticas.
6. O algoritmo implementado é híbrido: cada tipo de elemento define se o seu conteúdo será comparado utilizando o mecanismo de comparação sintática ou textual. Assim como apresentado nos resultados de Kim [KIM e NOTKIN, 2006], a utilização de uma comparação híbrida tende a encontrar resultados melhores.
7. O mecanismo de comparação implementado recebe apenas dois arquivos *A* e *B* (*two-way-diff*), porém com pouco esforço ele pode ser reutilizado para construir um mecanismo que receba também o arquivo base *C* (*three-way-diff*). Isso pode ser implementado da seguinte forma: o algoritmo atual realiza as comparações de *A* com *C* e *B* com *C* a fim de identificar os *deltas* de cada evolução, e utilizá-los como informação adicional para um novo algoritmo que receba *A, B*, *delta_1* e *delta_2*.
8. O modelo proposto não é capaz de detectar clones [GODFREY, 2005] (trechos de código replicados que foram inseridos ou removidos), em vez disso notifica operações de *Inserção* e *Remoção*, sem associá-las diretamente. Porém, com apoio de novas heurísticas é possível detectar também operações de unificação e separação (clones).