University of Passau
Department of Informatics and Mathematics

Master's Thesis

Applying Flexible Tree Matching to Abstract Syntax Trees

Author: Georg Seibt
`seibt@fim.uni-passau.de`

Supervisor: Prof. Dr.-Ing. Sven Apel
Olaf Leßenich

Submitted: 29/9/2016

Seibt, Georg:
Applying Flexible Tree Matching to Abstract Syntax Trees
Masterthesis, University of Passau, 2016.

Abstract

As software development projects grow, more and more merge scenarios between software artifacts occur. It is important to keep the number of conflicts as low as possible. Classical merge tools employ line-based algorithms to provide low runtime and applicability to all text documents.

The tool *JDime*[1] focuses on using the abstract syntax tree of the source code artifacts being merged to detect changes like moved methods that would produce conflicts in a line-based tool. The current tree matching algorithms however are, for performance reasons, constrained in the amount of matchings they consider.

This thesis builds on the work of Kumar et al. who introduced *Flexible Tree Matching* by extending their cost model for use with abstract syntax trees. The algorithm is implemented as a component of JDime with the option of using it as a replacement, post-processing step, or integrated part of the previously available tree matchers.

The matching quality of all three options is compared against the old matchers. For 5 previously difficult merge tasks, a significant increase in matching performance is observed. Post-processing provides the best compromise between increase in runtime and matching quality.

[1] URL: `http://www.infosun.fim.uni-passau.de/se/JDime/` (visited on 12/09/2016).

Contents

List of Figures

List of Tables

List of Algorithms

List of Listings

1 Introduction

The problem of having to merge multiple versions of a source code file quickly arises in larger software development projects. It is also a concern in product-line or model-driven software engineering. Version control programs like git[1] and Subversion[2] are used to handle these merge scenarios. They produce a merged version of the different revisions of the software artifacts that are involved in the merge scenario.

Several approaches exist for merging multiple revisions of a file into one. They may be categorized into textual, syntactic, semantic, and structural merging [Men02]. In this thesis, a general tree matching algorithm called *Flexible Tree Matching* is integrated into JDime alongside its classical tree matchers.

1.1 Motivation

Most available tools (such as the ones used by git and Subversion) perform unstructured (i.e. line based) merging. As such, the tools can resolve trivial merge conflicts but require user input even for situations that appear obvious to the user. However, they are applicable to all textual artifacts and the algorithms involved have a low runtime.

Some approaches to structured merging are implemented as part of the tool JDime. It supports merging the abstract syntax trees of Java source code files. JDime has previously been used to improve merge results over textual merging when methods are moved or code is reformatted [Leß12]. However, other changes, such as renaming of methods or surrounding of code with constructs like loops, could not be recognized due to the constraints inherent in the tree matching algorithms employed by JDime which will be described in Section 2.1.1.

Structured merging appears to be a good fit for preventing conflicts when source code is refactored in such a way that whole subtrees change position in the ASTs. This is the

[1] URL: `https://git-scm.com/` (visited on 05/09/2016).
[2] URL: `https://subversion.apache.org/` (visited on 05/09/2016).

case in the mentioned examples of surrounding a code fragment with a loop or conditional construct. If JDime can be improved to provide accurate matches between these ASTs, it may at some point represent a better alternative to the unstructured merge tools that are in use today.

1.2 Contributions

This thesis focuses on improving the tree matching capabilities of JDime by adding Flexible Tree Matching to its repertoire. The algorithm, described in detail in Chapter 3, uses a cost model based on weighted edge costs to express the quality of a set of matchings between two trees.

To make it applicable to the AST matching domain, a cost term dealing with ordering of children is added to the original cost model. The algorithm is further customized by making the weighing of costs dependant on the types of nodes being matched. In the current implementation, the renaming of method and class declarations is penalized less than that of other types of nodes. Additionally, the ordering cost is implemented such that it only applies when the ordering of the AST nodes is relevant. Weights, producing good matchings for situations that were previously hard to match, are learned using an evolutionary approach.

The resulting customized Flexible Tree Matching approach is combined in different ways with the old JDime matchers. To test its effectiveness, a number of merge tasks that are difficult to match using the previous algorithms are identified. Optimal reference matchings for these tasks are produced by hand. Using the new matcher as a post-processing step to try and match nodes that were left unmatched by the old matchers leads to a significant increase in matching performance. Runtime is also increased but not as much as by using Flexible Tree Matching as a replacement for the old algorithms.

1.3 Thesis Overview

In chapter Chapter 2, the general problem of graph isomorphism and its specialisation for trees is described. Two tools performing structured merging of source code as well as two papers improving upon aspects of structured merging are then introduced.

Chapter 3 recounts the definition of Flexible Tree Matching from the original paper and describes the extensions and modifications that were made to the algorithm. Chapter 4 then describes the implementation of Flexible Tree Matching as a component of JDime.

To evaluate the usefulness of Flexible Tree Matching in the AST merging domain, the optimal reference matchings for a number of difficult code examples are compared to those produced using the old matching techniques and the ones resulting from Flexible Tree Matching in Chapter 5.

2 Background

The most general form of the problem one faces when structurally merging abstract syntax trees is that of the isomorphism of graphs. Given two graphs G and H it is the problem of finding a bijection f between the vertices of G and H given by $V(\cdot)$. This function $f : V(G) \to V(H)$ has to have the property that an edge exists between two vertices a and b from G if and only if there exists one between $f(a)$ and $f(b)$. If such an isomorphism exists, the graphs are called isomorphic.

This problem is further specialized by adding labels (and possibly other attributes) to the vertices of the graphs and requiring that the bijection preserves all attributes of the vertices. In this general form, the problem is known to be in $\mathcal{NP}$, but it is neither known to be in $\mathcal{P}$ nor whether it is $\mathcal{NP}$-complete [For96].

In the structural merging domain, the problem is to find a set of matchings between two labelled trees. It is defined as a subset $M \subset L \times R$ where L and R are labelled trees. This subset has to have the property that no node from L or R occurs in more than one element of M. The elements of M can be taken to represent editing operations that transform L into R. An unmatched node in L is a deletion, one in R is an addition and nodes being matched identifies them as the same in both revisions.

Even though domain specific restrictions, such as ordering being important for many levels of an AST, can be made, finding such an optimal set of matchings still requires solving the largest common embedded subtree problem. This problem is known to be $\mathcal{APX}$-hard. As such, a polynomial-time approximation algorithm for the problem is possible.

2.1 Existing Tools

All widely used existing tools for merging software (such as GNU **diff**[1]) are line based. This is mainly due to line based merging being fast and applicable to all plain text

[1] URL: `https://www.gnu.org/software/diffutils/` (visited on 05/09/2016).

documents.

Structured merge tools promise less need for user interaction when resolving conflicts by using the structure inherent in code files. They work on the AST of the code file, not its lines.

Several research projects exist that attempt to use structured merging techniques to reduce the number of conflicts a user has to resolve by hand.

2.1.1 JDime

JDime is a structured merge tool being developed at the University of Passau, Germany. Since its introduction in the Master's thesis entitled "Adjustable Syntactic Merge of Java Programs" [Leß12] by Olaf Leßenich, it has been used in several projects dealing with structured merging.

It focusses on providing a structured merge that can resolve more conflicts than an unstructured merge, but still retains a similar runtime. In the evaluation done as part of the aforementioned Master's thesis, JDime proved to be a viable alternative to traditional line based merging when reordering or reformatting of code was involved in the merge scenario. By using an auto-tuning approach that first attempts a line based merge and then a structured merge if there are conflicts, the runtime of JDime is kept below that of purely structured tools.

While merging of reordered or reformatted code can be greatly improved by JDime, it lacks the capability to merge situations like code being surrounded by constructs such as loops or conditional statements. This is due to the tree matching algorithms in JDime being too restricted to be able to match the necessary parts of the ASTs. This thesis aims to improve the matchings in these situations.

2.1.2 ChangeDistiller

ChangeDistiller is a plugin for the Eclipse[2] Java IDE. It was developed by Fluri et al. in their paper entitled "Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction" [Flu+07]. The plugin uses the version control capabilities provided by the Eclipse platform to extract the version history of project files. It then extracts changes from successive revisions by matching the ASTs against each other.

[2] URL: `https://eclipse.org/` (visited on 26/09/2016).

The tree matching algorithm in ChangeDistiller is based on a more general algorithm for detecting changes in hierarchically structured data by Chawathe et al. [Cha+96]. It was customized to take the additional attributes and requirements of AST matching into account. The original algorithm produced sub-optimal matchings since it made assumptions that do not necessarily hold for ASTs. An example is the assumption that for any leaf in one tree, there exists at most one leaf in the other that is similar (or equal) to it. However, source code frequently contains similar statements.

The authors manually classified 1064 changes in 219 revisions of eight methods from three open source projects. Their algorithm proved to be effective in approximating the minimum edit script better than the original change extraction algorithm by Chawathe et al. did. The extracted changes are stored in a database. This repository of data about software changes can be used to answer questions about the kinds of changes that cause bugs when developing software.

2.1.3 Integrating into Version Control Systems

Most users of merging tools use them as part of larger version control systems. Major tools like git[3] and Subversion[4] include unstructured (i.e. line based) merge tools.

Both git and Subversion support using other merge tools that conform to a specific call syntax. Several applications make use of this facility to add graphical user interfaces for conflict resolution. Popular command line editors like Emacs and Vim also provide merge tools that can be used with git.

Another approach would be to use a structured merge tool such as JDime. In this way, the powerful version control systems could be extended with the improved conflict resolution that is possible when merging structurally. To enable this, JDime supports the call syntax expected by `git mergetool`[5].

2.2 Related Work

In this section two of the papers that deal with problems related to the matching of trees are mentioned. Section 2.2.1 describes an approach for calculating the size of the maximum common embedded subtree for ordered trees. Section 2.2.2 deals with a paper

[3] URL: `https://git-scm.com/` (visited on 05/09/2016).
[4] URL: `https://subversion.apache.org/` (visited on 05/09/2016).
[5] URL: `https://git-scm.com/docs/git-mergetool` (visited on 27/09/2016).

that aims to improve matching quality via a number of pre- and post-processing steps that can be applied to any matching algorithm.

2.2.1 Maximum Common Embedded Subtree in Ordered Trees

Lozano et al. reduce the maximum common embedded subtree problem for ordered trees to a variant of the longest common subsequence problem in their 2004 paper [LV04]. They present a dynamic programming algorithm that runs in polynomial time, namely $\mathcal{O}(n_1 \cdot n_2 \cdot \min(d_1, l_1) \cdot \min(d_2, l_2)))$. The trees being examined have n_1 and n_2 nodes, are of depth d_1 and d_2, and have l_1 and l_2 leaves.

They define the concept of a "Balanced Sequence" to describe trees as well-formed parenthesis strings over the alphabet $\{0, 1\}$. The balanced sequence of a leaf is empty, the balanced sequence of any other node is the concatenation of the balanced sequences of its children, each preceded by a 0 and followed by a 1. Additionally, they describe how to partition a balanced sequence s into its *head* and *tail*. They are the two unique subsequences such that $s = 0\ \mathit{head}(s)\ 1\ \mathit{tail}(s)$.

In Theorem 2 of the paper it is proved that the longest common balanced subsequence of two trees is the balanced sequence of a maximum common embedded subtree of the trees. The authors then give a recursive definition for the length of such a longest common balanced subsequence between two trees s and t.

$$
\begin{aligned}
&\mathit{lcs}(s, \lambda) = 0\\
&\mathit{lcs}(\lambda, t) = 0\\
&\mathit{lcs}(s, t) = \mathit{max}\begin{cases}\mathit{lcs}(\mathit{head}(s), \mathit{head}(t)) + \mathit{lcs}(\mathit{tail}(s), \mathit{tail}(t)) + 1\\ \mathit{lcs}(\mathit{head}(s)\ \mathit{tail}(s), t)\\ \mathit{lcs}(s, \mathit{head}(t)\ \mathit{tail}(t))\end{cases}
\end{aligned}
$$

The empty balanced sequence is denoted by λ. An implementation of the above definition would dynamically cache lengths for balanced sequences while the recursion is in progress.

This algorithm can be used for generating scores for matchings between subtrees as they are needed by the matcher implementation in JDime (see Section 4.2).

2.2.2 Move-Optimized Source Code Tree Differencing

In their 2016 paper Dotzler et al. contribute five general purpose optimizations for tree matching algorithms [DP16]. These optimizations are aimed at reducing the number of edit actions needed to express the changes between revisions of a source code file.

The optimization consist of one pre-processing step and four optimizations that may be applied after a tree matching algorithm produced a set of matchings. The pre-processing step is what the authors call "Identical Subtree Optimization". A fingerprint is calculated for every subtree in both trees being matched. One of the trees is then traversed and if the fingerprint of a subtree can be found in the other tree (and the fingerprints are unique in their ASTs) matchings between all nodes of the subtrees are recorded.

One example of a post-processing step is the so called "LCS Optimization". After the actual matching algorithm ran, the matchings are examined by flattening the subtrees whose roots were matched and applying the longest common subsequence (LCS) algorithm to them. If there are pairs of unmapped nodes with the same label in the longest common subsequence, the optimization step adds matchings between them.

The authors applied their optimizations to the tree matching algorithms used by three different tools and found that the edit scripts could be shortened in all cases. JDime includes a similar pre-processing step in the `EqualityMatcher` (see Section 4.2) but lacks the post-processing steps. Overall JDime uses a more general approach to tree matching, no optimizations or error corrections are performed after the matching in Algorithm 2 is completed.

3 Flexible Tree Matching

A matching between two labeled trees L and R can be viewed as a bipartite graph between the nodes of the trees. A classical measurement of the quality of such a matching is the tree edit distance [Bil05]. The matchings are interpreted as operations transforming L into R. A matching between nodes with a different label represents a renaming. Not matching a node in L or R represents deletion and insertion respectively. Given a cost function for every kind of operation one can define the tree edit distance as the minimum sum of costs of an edit script transforming L into R.

In the 2011 paper submitted to the IJCAI[1] by Kumar et al. the authors describe their approach to the tree matching problem [Kum+11b]. They define a set of matchings M as a subset of the complete bipartite graph G between the nodes of $L \cup \{\otimes_L\}$ and $R \cup \{\otimes_R\}$ with $\otimes_L$ and $\otimes_R$ representing additional *no-match* nodes. M must be chosen such that every node from $L \cup R$ is included in exactly one matching.

Classical tree matching algorithms have rigid requirements such as preservation of ancestry (matching nodes l and r implies that descendants of l may only be matched to those of r) and ordering of children. *Flexible Tree Matching* relaxes these requirements and introduces a per-matching cost model that makes it possible to weigh the role of different attributes of M to tune the results of the algorithm for approximating optimal matchings described in Section 3.3.

Given a function $c(m)$ calculating the cost of a matching $m \in M$, the problem of tree matching is finding a set of matchings that minimizes the sum of the costs defined as

$$c(M) = \frac{1}{|L| + |R|} \sum_{m \in M} c(m)\,.$$

In Section 5 of their paper Kumar et al. present a proof of the $\mathcal{NP}$-completeness of the decision problem "Does there exist a zero-cost flexible mapping between the trees?". They formulate a polynomial-time reduction from 3-PARTITION [GJ75]. As therefore no

[1] URL: `http://ijcai.org/` (visited on 05/09/2016).

polynomial-time algorithm for flexible tree matching can exist, the authors propose the stochastic algorithm described in Section 3.3 of this document to approximate the optimal set of matchings.

3.1 Cost Functions

To calculate $c(m)$ the authors use four weights which are applied to quantities (costs) describing properties of a matching. For $m = [l, r]$ they define $c(m)$ as

$$c([l,r]) = \begin{cases} w_n & \text{if } l = \otimes_L \vee r = \otimes_R \\ c_r([l,r]) + c_a([l,r]) + c_s([l,r]) & \textit{otherwise} \end{cases} .$$

The additional *no-match* nodes $\otimes_L$ and $\otimes_R$ were introduced to allow weighing the cost of not matching a node using the weight w_n. If m represents an actual match, three costs c_r, c_a and c_s, which are internally weighed using the corresponding w_r, w_a and w_s weights, are summed up.

The first summand $c_r(m)$ represents the cost of matching two nodes that have different labels. The cost is defined to be 0 if the labels of l and r match, w_r otherwise. Note that this is the only cost function not requiring knowledge about the other elements of M to be calculated.

The cost of violating ancestry relationships between the nodes in L and R is represented by c_a. This cost function examines the children of l and r to count the ones not being matched to children of the opposite node. Let $C(n)$ represent the children of a node and $M(n)$ the node that n is being matched with in the matchings M. The children violating the ancestry relationship are then given by

$$V_M(l) = \{l' \in C(l) \mid M(l') \in R \setminus C(M(l))\}$$

for a node in L and symmetrically for a node in R. Then we have

$$c_a([l,r]; M) = w_a \cdot (|V_M(l)| + |V_M(r)|) .$$

The cost c_s penalizes matchings that do not preserve sibling relationships between the trees. To calculate c_s, the functions $P(n)$ giving the parent of a node and $S(n) = C(P(n))$ (the siblings of a node) are introduced. The authors then define the sets of sibling-invariant, and sibling-divergent nodes. The former is the set of nodes in l's sibling group that are mapped to nodes in $M(l)$'s sibling group.

$$I_M(l) = \{l' \in S(l) \mid M(l') \in S(M(l))\}$$

The latter is the opposite set of nodes in l's sibling group that are not sibling-invariant but do represent an actual match.

$$D_M(l) = \{l' \in S(l) \setminus I_M(l) \mid M(l') \neq \otimes_R\}$$

Lastly the set of distinct parent families is given by

$$F_M(l) = \bigcup_{l' \in S(l)} P(M(l')) .$$

All three terms are defined symmetrically for r. The sibling group breakup cost is then calculated as

$$c_s([l, r]; M) = w_s \cdot \left(\frac{|D_M(l)|}{|I_M(l)| \cdot |F_M(l)|} + \frac{|D_M(r)|}{|I_M(r)| \cdot |F_M(r)|} \right) .$$

3.2 Bounding the Cost Functions

The cost functions c_a and c_s and c_o (see Section 3.5.2) present a problem when searching for an optimal set of matchings directly. To calculate the cost of a single matching, they expect the other matchings to be present in the previously defined form, having exactly one matching covering every node in the trees L and R. They can therefore not be used for directly calculating an optimal set of matchings.

Kumar et al. describe an approach for bounding the values of c_a and c_s when presented with a set $G \subset L \cup \{\otimes_L\} \times R \cup \{\otimes_R\}$ that may contain multiple elements covering a node from L or R. The bounds are determined by two questions: "Is it *possible* that the cost is incurred?" and "Is it *unavoidable* that the matching is penalized?". The former informs the upper bound, the latter the lower.

For c_a two indicator functions are introduced which encode these questions. Whether it is possible that a child will violate ancestry is defined as

$$1_a^{\mathcal{U}}(l', r) = \begin{cases} 1 & \text{if } \exists [l', r'] \in G : r' \notin C(r) \cup \{\otimes_R\} \\ 0 & \textit{otherwise} \end{cases} .$$

Whether an ancestry violation is guaranteed for a child l' is given as

$$1_a^{\mathcal{L}}(l', r) = \begin{cases} 1 & \text{if } \neg\exists[l', r'] \in G : r' \in C(r) \cup \{\otimes_R\} \\ 0 & \textit{otherwise} \end{cases} .$$

These functions are defined symmetrically for children from the right tree being tested against parents from the left (with the only change being the use of $\otimes_L$ instead of $\otimes_R$). The authors then define the upper and lower bounds for $c_a([l, r]; M)$ as

$$\mathcal{U}_a([l, r]) = w_a \cdot \left(\sum_{l' \in C(l)} 1_a^{\mathcal{U}}(l', r) + \sum_{r' \in C(r)} 1_a^{\mathcal{U}}(r', l) \right),$$

and

$$\mathcal{L}_a([l, r]) = w_a \cdot \left(\sum_{l' \in C(l)} 1_a^{\mathcal{L}}(l', r) + \sum_{r' \in C(r)} 1_a^{\mathcal{L}}(r', l) \right).$$

To bound c_s, the sizes of all involved sets have to be bounded first. Define the other siblings of a node n as $\overline{S}(n) = S(n) \setminus \{n\}$. To bound the sizes of $D(\cdot)$, $I(\cdot)$ and $F(\cdot)$ for a matching $m = [l, r]$, the questions "Is it possible that a node is in the set?" and "Is a node guaranteed to be in the set?" are encoded as indicator functions. Considering a node $l' \in \overline{S}(l)$ the authors define them as

$$1_D^{\mathcal{U}}(l', r) = \begin{cases} 1 & \text{if } \exists[l', r'] \in G : r' \notin \overline{S}(r) \cup \{\otimes_R\} \\ 0 & \textit{otherwise} \end{cases} ,$$

$$1_D^{\mathcal{L}}(l', r) = \begin{cases} 1 & \text{if } \neg\exists[l', r'] \in G : r' \in \overline{S}(r) \cup \{\otimes_R\} \\ 0 & \textit{otherwise} \end{cases} ,$$

and

$$\mathcal{U}_D([l, r]) = \sum_{l' \in \overline{S}(l)} 1_D^{\mathcal{U}}(l', r),$$

$$\mathcal{L}_D([l, r]) = \sum_{l' \in \overline{S}(l)} 1_D^{\mathcal{L}}(l', r).$$

To bound the invariant sibling set size $|I(l)|$ similar indicators are defined:

$$1_I^{\mathcal{U}}(l', r) = \begin{cases} 1 & \text{if } \exists[l', r'] \in G : r' \in \overline{S}(r) \\ 0 & \textit{otherwise} \end{cases} ,$$

$$1_I^{\mathcal{L}}(l', r) = \begin{cases} 1 & \text{if } \forall [l', r'] \in G : r' \in \overline{S}(r) \\ 0 & \textit{otherwise} \end{cases} ,$$

and

$$\mathcal{U}_I([l, r]) = 1 + \sum_{l' \in \overline{S}(l)} 1_I^{\mathcal{U}}(l', r),$$

$$\mathcal{L}_I([l, r]) = 1 + \sum_{l' \in \overline{S}(l)} 1_I^{\mathcal{L}}(l', r).$$

As they appear in the denominator of the sibling cost calculation, the bounds for the number of distinct sibling families are only relevant if the numerator is nonzero. In this case the lower bound for $|F_M(l)|$ is 2 (as there is at least one divergent sibling having a different parent than l) and the upper bound is $\mathcal{L}_D(l, r) + 1$.

All other quantities are defined symmetrically. The bounds for $c_s([l, r]; M)$ are then given as

$$\mathcal{U}_s([l, r]) = \frac{w_s}{2} \cdot \left(\frac{\mathcal{U}_D(l, r)}{\mathcal{L}_I(l, r)} + \frac{\mathcal{U}_D(r, l)}{\mathcal{L}_I(r, l)} \right)$$

and

$$\mathcal{L}_s([l, r]) = w_s \cdot \left(\frac{\mathcal{L}_D(l, r)}{\mathcal{U}_I(l, r) \cdot (\mathcal{L}_D(l, r) + 1)} + \frac{\mathcal{L}_D(r, l)}{\mathcal{U}_I(r, l) \cdot (\mathcal{L}_D(r, l) + 1)} \right).$$

The bounds for the cost of a matching are then the sum of all bounds and the renaming cost c_r.

$$c_{\mathcal{U}}(m) = c_r(m) + \mathcal{U}_a(m) + \mathcal{U}_s(e)$$

$$c_{\mathcal{L}}(m) = c_r(m) + \mathcal{L}_a(m) + \mathcal{L}_s(m)$$

3.3 Approximating the Lowest Cost Matchings

The authors use the Metropolis algorithm to iteratively approximate an optimal set of matchings [CG95]. To that end they define their objective function as

$$f(M) = \exp[-\beta \cdot c(M)] .$$

The constant β is determined based on the size of the trees. It is responsible for scaling the costs of two sets of matchings, in effect making it more or less likely that a set with a higher cost will replace a comparatively cheaper set. Every iteration of the Metropolis algorithm proposes a new set of matchings $\hat{M}$. This set becomes the new reference with probability

$$\alpha(\hat{M} \mid M) = \min\left(1, \frac{f(\hat{M})}{f(M)}\right).$$

To initialize M the algorithm starts with the complete bipartite graph G and calculates the cost bounds of all matchings. Then G is ordered by increasing bound and traversed. Every matching is considered for assignment to M with a fixed probability p. After a matching is chosen, it is fixed in G and assigned to M. Any other matchings that contain a node from the fixed matching are then removed from G and the bounds of all matchings remaining in G are then recalculated. Since there is now less variability in the matchings, the new bounds can be calculated more accurately. This continues until only fixed matchings remain in G. M is then returned.

Proposing a new set of matchings $\hat{M}$ employs the same selection process used for initialization. G is set to the complete bipartite graph and a random index j from $[1, |M|]$ is chosen. The first j matchings in M are then fixed. The rest of the matching is arrived at using the same selection process as above.

Over all iterations, the set of matchings with the lowest cost is stored and returned in the end.

3.4 Application

In the original paper, the implementation of the Flexible Tree Matching algorithm is motivated by the goal of automatically retargeting Web pages [Kum+11b; Kum+11a]. The authors apply the algorithm to a segmentation of the DOM tree that represents the visual structure of the Web pages. The matching between Web pages is used to guide the transfer of content and design between them.

Kumar et al. tune the parameters of the cost model by having participants in a study specify matchings between 52 unique pairs of page trees. Learning a consistent cost model is complicated by the fact that humans do not produce identical mappings between pages. Nevertheless, the algorithm was successfully used for prototyping of websites. The content of an original website can be automatically retargeted to a number of different layouts and styles.

3.5 Extensions

As this thesis applies Flexible Tree Matching to abstract syntax trees, the original approach was extended in two main ways.

3.5.1 Weighting Functions

In the original paper the weighing of costs was performed by either multiplying some quantity produced in the calculation of the cost by a weight, or returning the weight itself as the cost. The former held for c_a and c_s while the latter was the case for c_r and not matching a node.

While the existing cost functions themselves were not modified, the weights were replaced by weighting functions. In addition to the quantity that is being weighed (in the case of c_a and c_s) the matching that is being evaluated is passed to the weighting function. It may then be implemented as simple multiplication, ignoring the additional parameter, but may also modify the given weight based on some attribute of the AST nodes being matched. In the implementation described in Chapter 4 the cost of renaming for nodes representing method or class declarations is lowered. Another use would be to eliminate the ordering cost introduced in Section 3.5.2 when the order of the nodes being matched is determined to be irrelevant.

3.5.2 Ordering Cost

Ordering is an important concern in ASTs. Though it may be unimportant for certain language elements (such as method declarations in Java), for most levels of the tree ordering is relevant. Keeping in theme with the original cost model, an additional cost term and weighting function was introduced. The function c_o and its weight w_o penalize matchings that introduce an ordering which is being violated by other matchings.

A test whether a matching $m_1 = [l_1, r_1]$ violates the ordering introduced by another matching $m_2 = [l_2, r_2]$ is needed to calculate c_o. To be able to compare all pairs of matchings, the function $LCA(x, y)$ is defined for nodes x and y from the same tree. The function returns the lowest pair of ancestors of x and y that are part of the same sibling group. Figure 3.1 shows the ancestors (marked by endpoints of red arrows) that would be returned for calling $LCA(B, C)$ and $LCA(A, C)$ respectively.

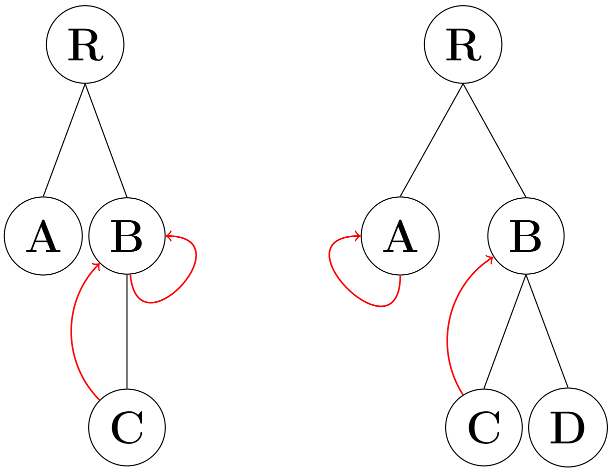

Figure 3.1: The Lowest Common Ancestors

Let $I(n)$ be the index of node n in its sibling group and $LCA(l_1, l_2) = (l_{a1}, l_{a2})$ and $LCA(r_1, r_2) = (r_{a1}, r_{a2})$. If the order of all nodes in the sibling groups of l_{a1} and r_{a1} is irrelevant, no ordering violation is possible. Otherwise the ordering violation test is defined as

$$VO([l_1, r_1], [l_2, r_2]) = \begin{cases} I(r_{a1}) > I(r_{a2}) & \text{if } I(l_{a1}) < I(l_{a2}) \\ I(r_{a1}) < I(r_{a2}) & \text{if } I(l_{a1}) > I(l_{a2}) \\ \mathit{false} & \mathit{otherwise} \end{cases} .$$

Given a matching $[l, r] \in M$ the matches containing siblings of l or r are examined. SM_M selects all relevant matches from M to compare them with the matching being evaluated.

$$SM_M([l, r]) = \{[l', r'] \in M \mid (l' \in \overline{S}(l) \wedge r' \in R) \vee (l' \in L \wedge r' \in \overline{S}(r))\}$$

The cost function c_o can then be defined as

$$c_o(m; M) = \begin{cases} w_o(m) & \exists m' \in SM_M(m) : VO(m', m) \\ 0 & \mathit{otherwise} \end{cases} .$$

Figure 3.2 shows a set of matchings between the trees rooted in L and R. None of the matchings represented by edges between the trees violate the ordering constraint imposed by c_o even though the nodes B', B'', C', and C'' were introduced.

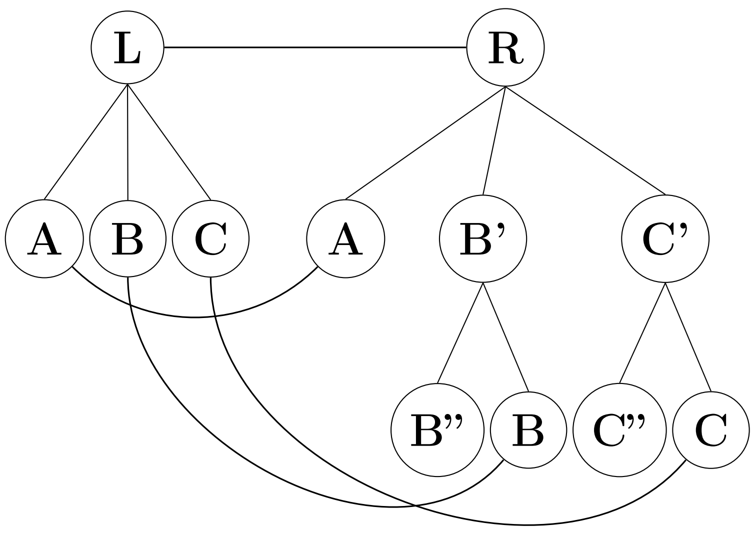

Figure 3.2: A Correctly Ordered Set of Matchings

To bound the cost $c_o(m; M)$ with $\mathcal{L}_o(m)$ and $\mathcal{U}_o(m)$ one has to answer two questions for all siblings of the nodes l and r contained in m. Is it possible that a matching containing one of the siblings will introduce an ordering violation? Secondly, is it unavoidable that such a violation exists? As with the original bounding cost functions, the former question determines the upper bound, and the latter the lower.

Let G be a subset of the complete bipartite graph between the nodes of L and R and $MS_G(n) = \{[l, r] \in G \mid l = n \vee r = n\}$ a function collecting all matchings containing n from G. Furthermore $NM([l, r]) = l = \otimes_L \vee r = \otimes_R$ determines whether a matching represents a no-match. We can then determine whether a violation is possible using

$$VP_G([l, r]) = \exists s \in \overline{S}(l) \cup \overline{S}(r) : \exists m \in MS_G(s) : \neg NM(m) \wedge VO(m, [l, r])$$

and whether ordering is possible with

$$OP_G([l, r]) = \forall s \in \overline{S}(l) \cup \overline{S}(r) : \exists m \in MS_G(s) : NM(m) \vee \neg VO(m, [l, r])\,.$$

If ordering is not possible, we get $\mathcal{L}_o(e) = \mathcal{U}_o(e) = w_o(e)$, otherwise we set $\mathcal{L}_o(e) = 0$ and $\mathcal{U}_o(e) = w_o(e)$ if a violation is possible, otherwise it is also zero.

4 Implementation

The tool JDime[1] was first introduced as part of the masters thesis by Olaf Leßenich in [Leß12]. Chapter 4 of that thesis described the implementation of the tool. The key goals of the design were adjustability and extensibility with the latter being particularly important for this thesis.

JDime has the ability to first run an unstructured merge and switch to a structured merge if the first attempt produced conflicts. Thereby the complexity of the merge strategy can be adjusted to the complexity of the merge itself. The application was also designed to make the implementation and integration of new algorithms relatively easy. This makes JDime the ideal environment for executing and evaluating Flexible Tree Matching for abstract syntax trees. The algorithm was therefore implemented as an additional tree-matcher for JDime with the hope of improving the existing matching results.

4.1 Basic Datastructures

JDime makes heavy use of the type generics and inheritance features of the Java programming language to enable merging of directory trees, and abstract syntax trees using essentially the same code path. It is important to understand a few basic data structures used in the program to get an overview of how a merge proceeds.

4.1.1 Artifact

The basic building block of the JDime architecture is the `Artifact` class. Most other classes are parametrized with the specific type of `Artifact` that they deal with. An `Artifact` represents (in the current JDime version) either a file in a directory tree or a node in an AST.

[1] URL: `http://www.infosun.fim.uni-passau.de/se/JDime/` (visited on 12/09/2016).

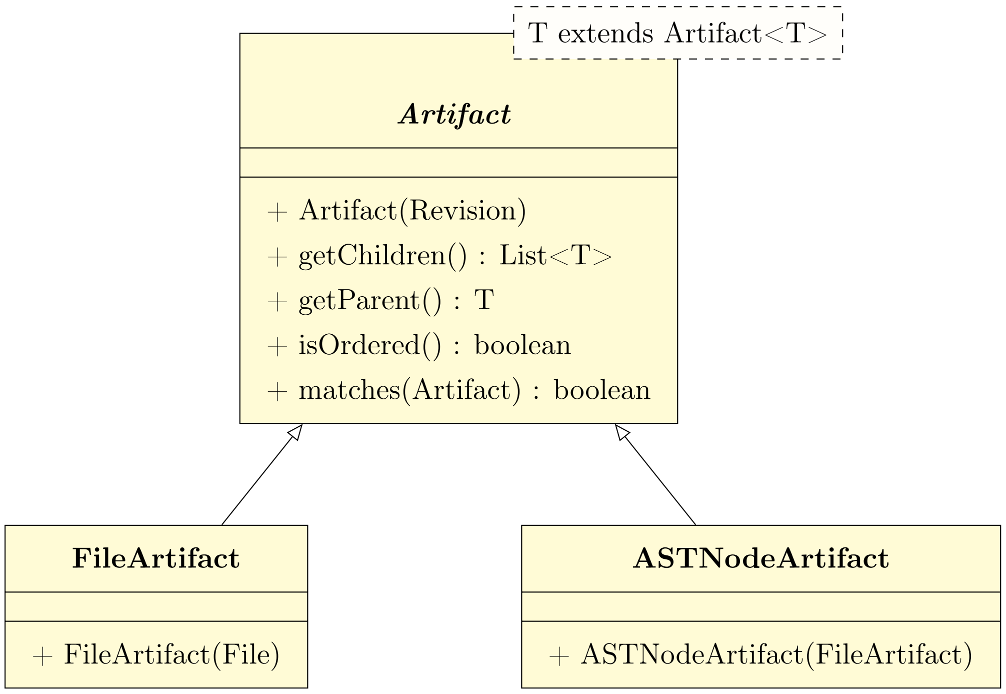

Figure 4.1: The `Artifact` class structure

The tree structure is implemented generically in the base `Artifact` class shown in Figure 4.1. Most other classes, such as `Merge`, work with any kind of `Artifact`. This opens up the possibility of using the same code for merging both directory trees and abstract syntax trees. If one were to extend JDime for a different language than Java, implementing a new `Artifact` representing a node in the new AST would be the first step.

To produce an `ASTNodeArtifact` from a `FileArtifact`, a Java AST builder is required. The decision was made to use ExtendJ[2] (formerly known as JastAddJ). ExtendJ is an extensible compiler that is based on the JastAdd[3] meta-compilation system. In ExtendJ an AST is built from nodes that are part of a Java class hierarchy, each representing a different piece of the abstract syntax. This AST is wrapped in `ASTNodeArtifact` instances.

For the `Matcher` that is described in Section 4.2 the methods `isOrdered()` and `matches(Artifact)` are particularly important. The former is used to determine whether the order of this artifact and its siblings is important when merging. For a concrete `ASTNodeArtifact` representing an `import` statement this method would return *false* since `import` statements may be given in any order in Java. On the other hand, the order of statements within a method body must be taken into consideration. The `matches(Artifact)`

[2]URL: `http://jastadd.org/web/extendj/` (visited on 12/09/2016).
[3]URL: `http://jastadd.org/web/` (visited on 12/09/2016).

method returns whether an `Artifact` can be considered equal to another `Artifact` irrespective of its children. One can consider the nodes of an artifact tree as being implicitly labeled by their `matches(Artifact)` function.

4.1.2 MergeStrategy

The strategy pattern is used in JDime to select the general approach to merging a set of inputs [Gam+96]. Using the `-mode MODE` command line option a strategy may be chosen. All extensions of the abstract `MergeStrategy` class implement the `MergeInterface` and can merge artifacts contained in a `MergeOperation` using the configuration options represented by the `MergeContext`. The three most used merge strategies are shown in Figure 4.2.

The `LineBasedStrategy` simply passes the given input files to `git merge-file`[4], collects the output and uses it as the result of the merge. This strategy is used on its own for sanity checking and to compare the runtime of a traditional unstructured merge to any other approaches implemented in JDime. It is also the first stage of the `CombinedStrategy`. Its main advantage is the low runtime when compared to the more complex algorithms employed by the `StructuredStrategy`.

When using a `StructuredStrategy`, the input `FileArtifact` instances are converted to `ASTNodeArtifact` trees. The strategy then employs the `Merge` class that is described in Section 4.1.3 to merge the trees. The resulting combined tree is pretty-printed back to source code and represents the result of the merge. Changes that would cause a conflict using unstructured merging, such as changing the order of method declarations or imports, can be detected by the `StructuredStrategy` [ALL12].

[4]URL: `https://git-scm.com/docs/git-merge-file` (visited on 12/09/2016).

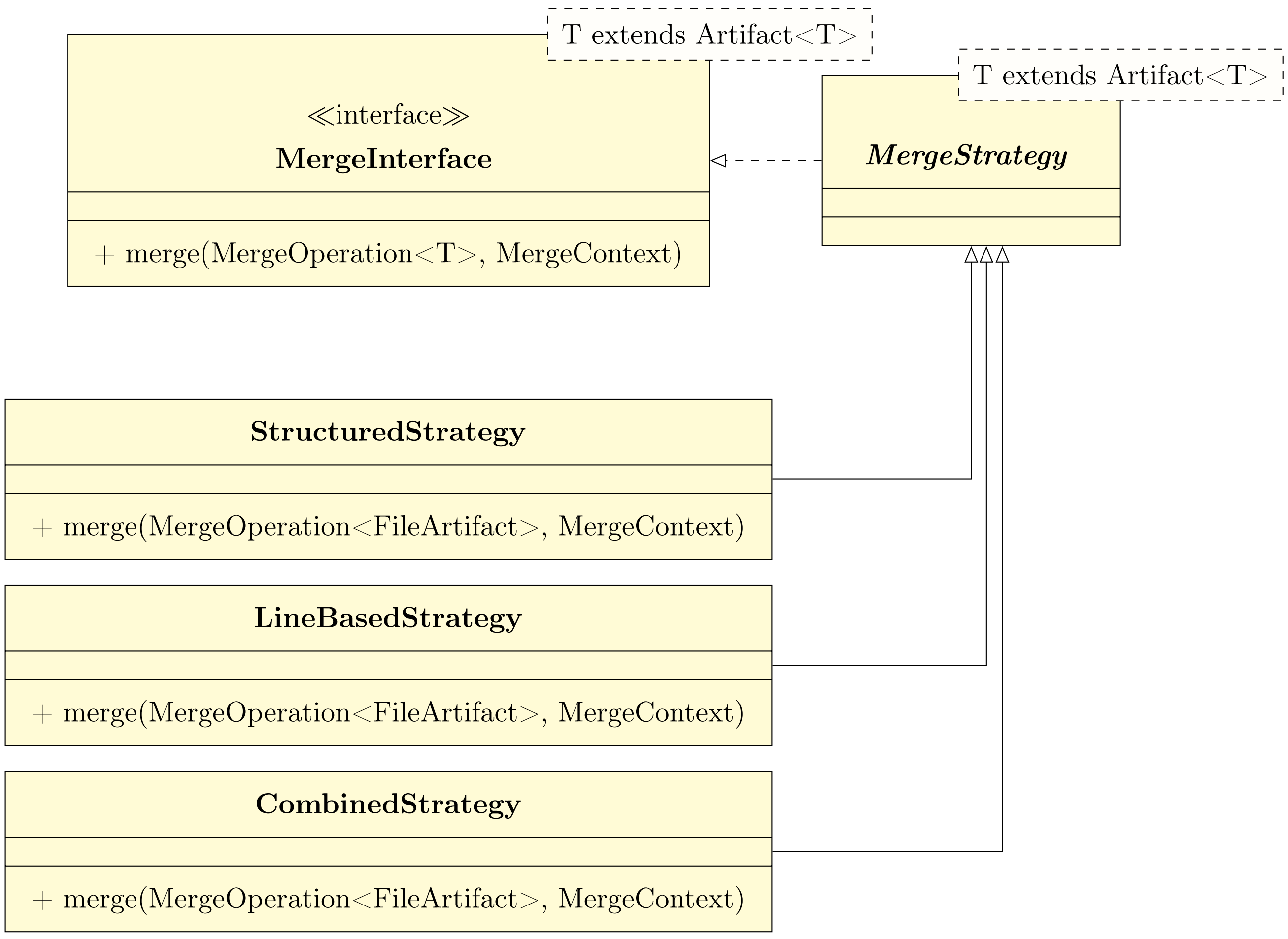
T extends Artifact<T>
≪interface≫
MergeInterface
+ merge(MergeOperation<T>, MergeContext)
T extends Artifact<T>
MergeStrategy
StructuredStrategy
+ merge(MergeOperation<FileArtifact>, MergeContext)
LineBasedStrategy
+ merge(MergeOperation<FileArtifact>, MergeContext)
CombinedStrategy
+ merge(MergeOperation<FileArtifact>, MergeContext)

Figure 4.2: The `MergeStrategy` class structure

The `CombinedStrategy` implements the approach described in Chapter 4 that combines unstructured and structured merging. First, a `LineBasedStrategy` is applied to check whether the files are mergeable using traditional methods. If the `LineBasedStrategy` produces a merge result containing conflicts, a structured merge is then attempted using the `StructuredStrategy`. By using the cheaper unstructured merge first, and only using the structured strategy if it is necessary, the `CombinedStrategy` achieves less conflicts than the `LineBasedStrategy` while being significantly faster than the `StructuredStrategy`. In an evaluation involving 72 merge scenarios with more than 17 million lines of code it was measured to be 5 times faster on average than purely structured merging [ALL12].

There are also other, more specialized `MergeStrategy` implementations available. The `NWayStrategy` for example implements a structured merge between an arbitrary number of input files. A semistructured merge that uses unstructured merging for method bodies but structured merging for the AST above the methods has also been implemented in JDime as a proof of concept [Ape+11]. A full reference implementation, using superimposition of program structure trees, is included in FeatureHouse[5].

4.1.3 Merge

The class `Merge` is used to merge `Artifact` trees. Depending on the number of input artifacts either a three-way or a two-way merge is performed. In any case, the `Matcher` that is described in detail in Section 4.2 is used to compare the necessary pairs of artifact trees.

Algorithm 1: The General Merge Algorithm

Input: The artifact trees being merged as L, B, R
Output: The combined artifact tree

```
1  if $B \neq null$ then
2      Match($L$, $B$)
3      Match($B$, $R$)
4  end
5  Match($L$, $R$)
6  if AnyChildOrdered($L$, $B$, $R$) then
7      return OrderedMerge($L$, $B$, $R$)
8  else
9      return UnorderedMerge($L$, $B$, $R$)
10 end
```

The call to `Match` determines the matchings between the trees and stores information about the matching partner in the nodes of the trees. Then the children of L, B, and R are examined to determine whether any of them are ordered. If that is the case, an `OrderedMerge` is performed, otherwise an `UnorderedMerge` is used. In both cases the concrete merge algorithm considers the direct children of the trees and executes add-, delete-, merge-, or conflict-operations depending on the matchings. To merge the children's children the concrete merge functions again call the general merge function to determine the correct approach and execute the merge.

4.2 Matcher

In the current implementation of the `Matcher` class that provides the `Match` method used in Algorithm 1, a strategy similar to the one in the `Merge` class is used. The matching

[5] URL: `http://www.infosun.fim.uni-passau.de/spl/apel/fh/` (visited on 15/09/2016).

algorithm is implemented top down using a general method that, based on the attributes of the artifacts being matched, decides which concrete matching algorithm to use. The class structure surrounding the `Matcher` is shown in Figure 4.3.

To optimize the case where L and R are exactly the same, the `EqualityMatcher` is used to provide the `TrivialMatch` function. Therein the trees are traversed in depth first order and checked for exact equality. The `EqualityMatcher` will only return a set of matchings if there are no differences between the trees. The complexity of the `EqualityMatcher` is $\mathcal{O}(n)$ where n is the number of nodes in the smaller of both trees. In contrast, the general matching may employ matchers like the `HungarianMatcher` which has a $\mathcal{O}(n^3)$ runtime where n is the maximum number of nodes in one of the trees being matched.

Algorithm 2: The General Match Algorithm

Input: The artifact trees being matched as L, R

Output: The matchings between nodes of L and R

```
matchings ← TrivialMatch($L$, $R$)
if matchings ≠ $null$ then
    return matchings
end
if Matches($L$, $R$) then
    if AnyChildOrdered($L$, $R$) then
        matchings ← OrderedMatch($L$, $R$)
    else
        matchings ← UnorderedMatch($L$, $R$)
    end
end
return matchings
```

If they do not trivially match, L and R are then compared using their `matches(Artifact)` method. If they do not match, the algorithm returns no matchings, otherwise an appropriate strategy is chosen to produce a match between L and R based on a matching between their children. The match will have an associated score indicating how many of the children could be matched. All concrete implementations behind `OrderedMatch` and `UnorderedMatch` need to match pairs of children of L and R to arrive at a final match between the artifacts. For this purpose the general `Match` method is used again.

For the unordered case, currently two implementations of `UnorderedMatch` exist. Firstly, the `HungarianMatcher`, which constructs a matrix of matchings between all pairs of

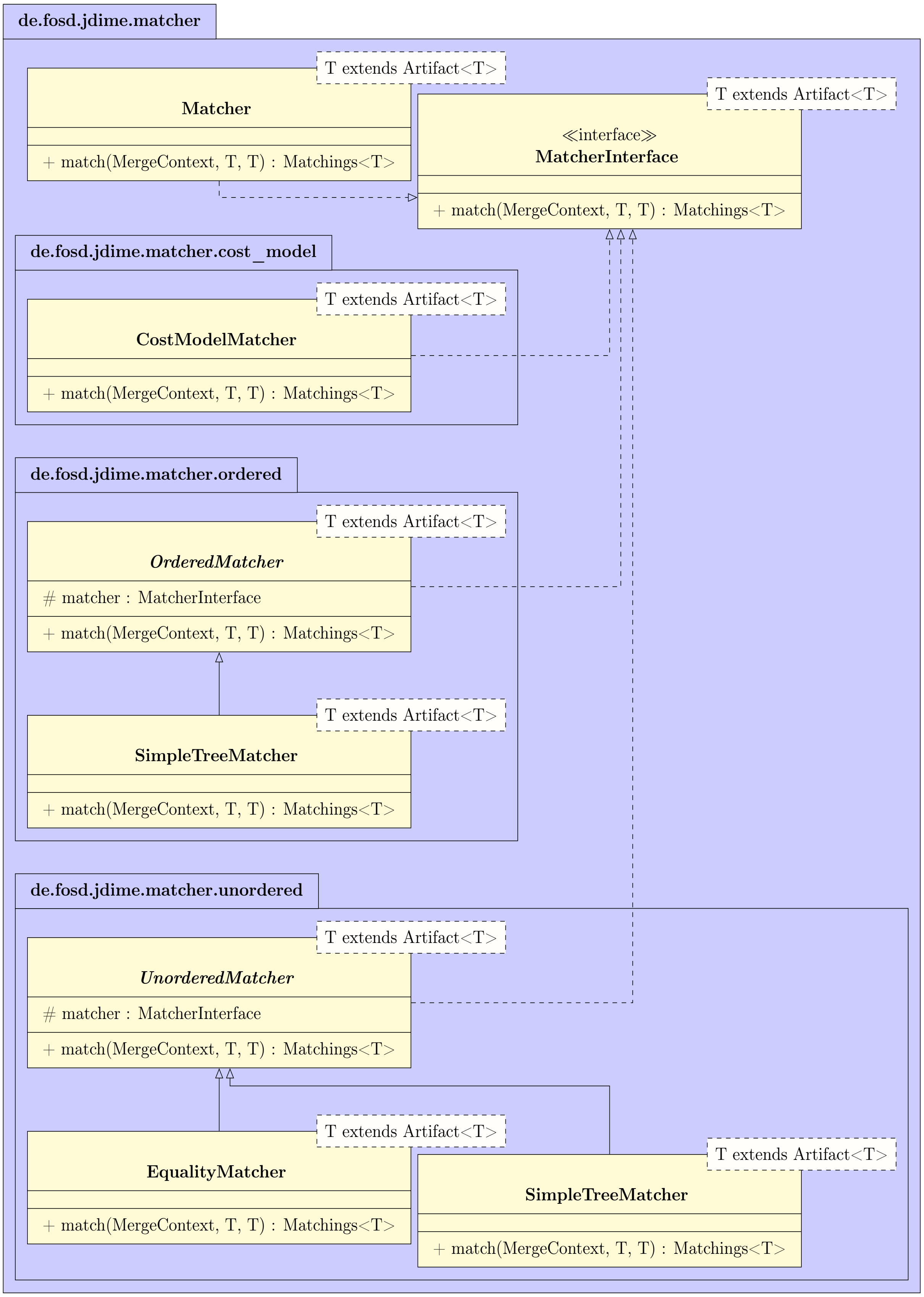
de.fosd.jdime.matcher
T extends Artifact<T>
Matcher
+ match(MergeContext, T, T) : Matchings<T>
T extends Artifact<T>
≪interface≫
MatcherInterface
+ match(MergeContext, T, T) : Matchings<T>
de.fosd.jdime.matcher.cost_model
T extends Artifact<T>
CostModelMatcher
+ match(MergeContext, T, T) : Matchings<T>
de.fosd.jdime.matcher.ordered
T extends Artifact<T>
OrderedMatcher
# matcher : MatcherInterface
+ match(MergeContext, T, T) : Matchings<T>
T extends Artifact<T>
SimpleTreeMatcher
+ match(MergeContext, T, T) : Matchings<T>
de.fosd.jdime.matcher.unordered
T extends Artifact<T>
UnorderedMatcher
# matcher : MatcherInterface
+ match(MergeContext, T, T) : Matchings<T>
T extends Artifact<T>
EqualityMatcher
+ match(MergeContext, T, T) : Matchings<T>
T extends Artifact<T>
SimpleTreeMatcher
+ match(MergeContext, T, T) : Matchings<T>

Figure 4.3: The `Matcher` class structure

children of L and R, interprets the scores of the matchings as edge weights in a bipartite graph between the two sets of children and then solves the assignment problem using the Hungarian Method as described in [Kuh10].

If all nodes being matched have a string representation that is unique in their sibling group (as is the case for example for `import` statements in the ExtendJ AST) a `UniqueLabelMatcher` is used. This matcher sorts the children of both L and R by their unique label and then iterates once over both lists simultaneously to find pairs of matching children.

For ordered trees the `SimpleTreeMatcher` is used. The class implements a customized version of Yang's "Simple Tree Matching" [Yan91]. Yang generalized a longest common subsequence algorithm to the problem of matching two trees. A matrix of similarity scores between children of L and R is constructed by recursively using the Simple Tree Matching algorithm on pairs of children. In JDime the recursive call is replaced by a call to the general `Match` method from Algorithm 2. The runtime complexity of the original implementation is in $\mathcal{O}(m \cdot n)$ where m and n are the sizes of the trees L and R.

4.2.1 Limitations

The `Match` method was designed to give an acceptable runtime without using heuristics to match the trees. It solves the *Largest Common Subtree*-problem as opposed to the *Largest Common **Embedded** Subtree*-problem. The former can be solved in $\mathcal{P}$ for both ordered and unordered trees, while the latter is known to be $\mathcal{NP}$-hard for unordered trees but is also in $\mathcal{P}$ in the ordered case [ZSS92]. Unordered trees are the norm when working with abstract syntax trees. Inherent in the design of the `Match` method are two limitations which this thesis aims to improve upon.

If two nodes do not match, the algorithm aborts instead of comparing the subtrees. Matchings between children of L and R are never considered. Let L and R represent the declaration of the same method (in different revisions of a source code file) and their children the statements within the method. If the only change between the revision were the renaming of that method, then L and R themselves would no longer match, but all their children would be exactly the same. In the current implementation of the `Match` method, the renaming would not be detected and a conflict between the two methods would be produced.

Due to the concrete `MatcherInterface` implementations being used, only artifacts on the same level in the overall ASTs can be matched. Certain plausible changes in the source code, such as surrounding a number of lines with an `if` statement, however lead to a portion of the AST moving down one or more levels. The nodes that were moved down can not be matched to their counterparts in the other revision as they are no longer on the same level.

Given well-formatted source code files, an unstructured merge would handle these cases better than the current structured approach. The lines representing the method body (in the renaming case) and the shifted code would be matched by the unstructured algorithm. Producing useful matchings in these situations using a structured strategy is the current development focus of JDime and the motivation for implementing *Flexible Tree Matching* as an additional matching method.

4.3 Implementing Flexible Tree Matching

To extend the JDime matching capabilities a new class implementing the `MatcherInterface` was added. The `CostModelMatcher` uses the cost model introduced in Chapter 3 and the Metropolis algorithm to approximate an optimal set of matchings between AST nodes [Kum+11b; CG95].

The main goal in adding the `CostModelMatcher` was to develop a way of generically dealing with situations that the other matchers could not. In Flexible Tree Matching, all possible pairs of nodes from the two trees may be considered as a matching. This makes it possible to detect renaming of methods and classes as well as code that has been shifted in one tree. The downside of this generalized approach is that less strict guarantees can be made about the matchings. Ordering of siblings is the most striking example: the algorithm can only be incentivized to respect the ordering in the matchings it chooses.

Flexible Tree Matching was introduced independently of implementation concerns in Chapter 3. To achieve an acceptable runtime, several optimizations had to be made in the implementation as a JDime matcher. As the algorithm does not share the limitations described in Section 4.2.1 an increase in runtime over the previous combination of matching algorithms was to be expected. The quality of matchings and the runtime of the algorithm will be further discussed in Chapter 5.

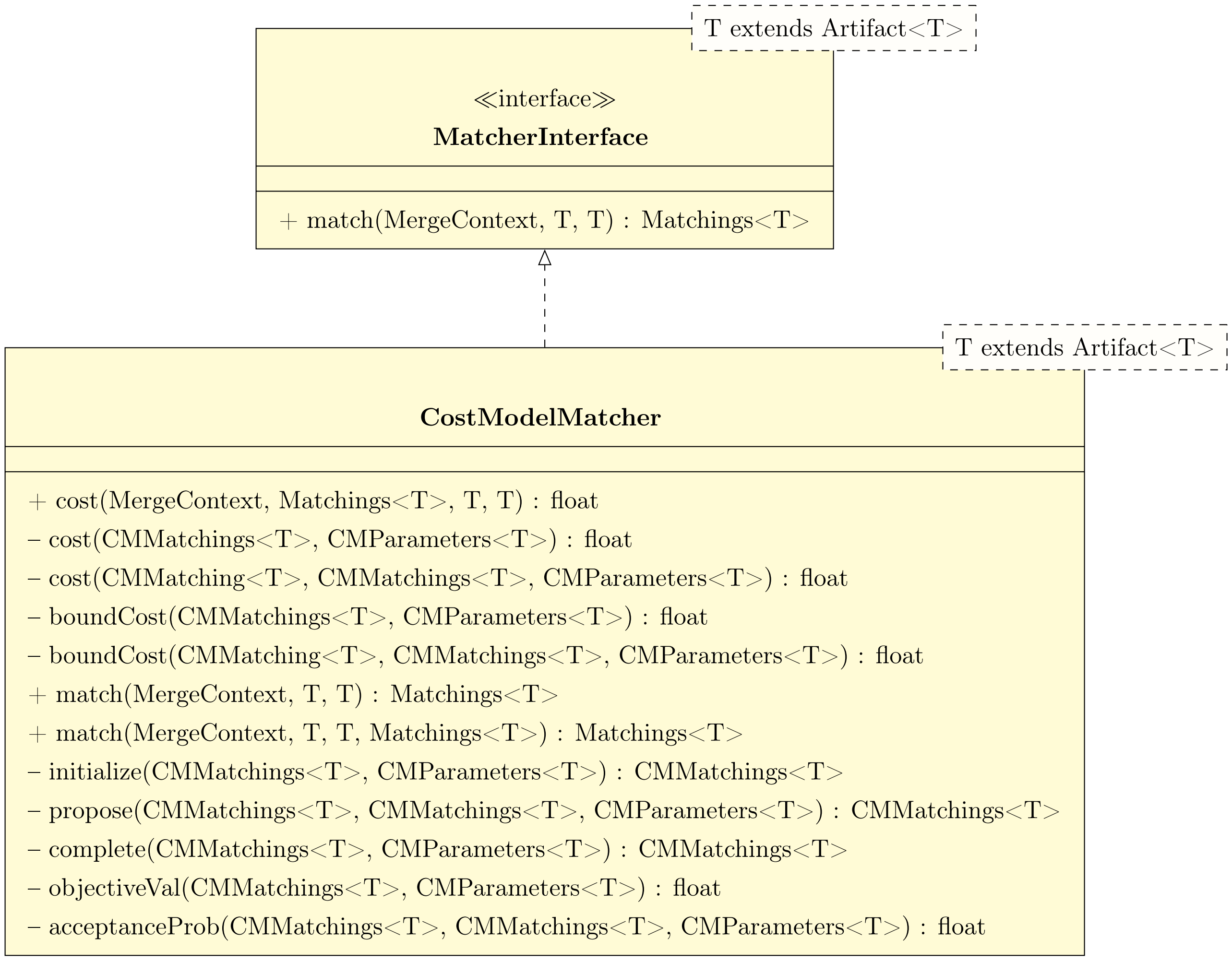
T extends Artifact<T>
≪interface≫
MatcherInterface
+ match(MergeContext, T, T) : Matchings<T>
T extends Artifact<T>
CostModelMatcher
+ cost(MergeContext, Matchings<T>, T, T) : float
– cost(CMMatchings<T>, CMParameters<T>) : float
– cost(CMMatching<T>, CMMatchings<T>, CMParameters<T>) : float
– boundCost(CMMatchings<T>, CMParameters<T>) : float
– boundCost(CMMatching<T>, CMMatchings<T>, CMParameters<T>) : float
+ match(MergeContext, T, T) : Matchings<T>
+ match(MergeContext, T, T, Matchings<T>) : Matchings<T>
– initialize(CMMatchings<T>, CMParameters<T>) : CMMatchings<T>
– propose(CMMatchings<T>, CMMatchings<T>, CMParameters<T>) : CMMatchings<T>
– complete(CMMatchings<T>, CMParameters<T>) : CMMatchings<T>
– objectiveVal(CMMatchings<T>, CMParameters<T>) : float
– acceptanceProb(CMMatchings<T>, CMMatchings<T>, CMParameters<T>) : float

Figure 4.4: The `CostModelMatcher` class.

The most important methods used to implement Flexible Tree Matching as a `MatcherInterface` are shown in Figure 4.4. The class provides three public methods. The `match(MergeContext, T, T)` method is mandated by the interface and returns a set of matchings between nodes of the two given trees. An overloaded version of this method adds an additional parameter to specify a set of matchings that were fixed beforehand. This version of the `match(MergeContext, T, T)` method will only add matchings between nodes that were not matched in the previously fixed matchings. This detail has been elided from the following pseudocode algorithms to improve readability. `cost(MergeContext, Matchings<T>, T, T)` is provided to easily calculate the cost of a set of matchings. This is used primarily for comparing the results of classical matchers to those of the `CostModelMatcher`.

The cost model itself is implemented as two sets of functions which are used internally by the `match(...)` method. The private version of `cost(...)`, shown as pseudocode in

Algorithm 3: The Exact Cost Functions

```
Function Cost(Matchings) is
    if IsEmpty(Matchings) then
        return 0
    end
    sum ← 0
    for M ← Matchings do
        sum ← sum + Cost(M, Matchings)
    end
    lSize ← TreeSize(Matchings.L)
    rSize ← TreeSize(Matchings.R)
    return sum * $\frac{1}{\text{lSize}+\text{rSize}}$
end
Function Cost(Matching, Matchings) is
    if IsNoMatch(Matching) then
        return NoMatchCost(Matching)
    end
    $c_r$ ← RenamingCost(Matching, Matchings)
    $c_a$ ← AncestryViolationCost(Matching, Matchings)
    $c_s$ ← SiblingGroupBreakupCost(Matching, Matchings)
    $c_o$ ← OrderingCost(Matching, Matchings)
    return $c_r + c_a + c_s + c_o$
end
```

Algorithm 3, calculates the exact cost of a set of matchings based on the currently set weights. It expects that for every node of the two trees, exactly one matching including this node exists. The `boundCost(...)` function is implemented analogously to Algorithm 3. It however assumes that there may be more than one matching containing a given node from one of the trees and therefore can only determine a lower and upper bound for c_a, c_s and c_o.

It should also be noted that the specific ordering cost function from Section 3.5 has been added to both the exact and bounded cost functions.

The specific cost functions also implement the modification to the weighting mechanism introduced in Section 3.5.1. When a quantity is being weighed it is not simply multiplied by a weight but passed (along with the matching for which the cost is being computed) to a weighting function. This function may be implemented as multiplication but it may also be more complex. The weighting function for the cost of renaming is implemented in such a way that method and class declarations are cheap to rename.

Two main measures were taken to improve the practical runtime of the algorithm. As the calculation of the exact, as well as the bounded cost of an edge only requires read access to the state of the `CostModelMatcher`, both the `cost(...)` and `boundCost(...)` functions are executed in parallel for every matching in the set that is being evaluated. Thanks to the `Stream` class that was introduced in Java 8, this feature was easily implemented. Additionally a cache was used to store the results of some computations that are needed by several of the specific cost functions. In the case of the `boundCost` function both the c_a and c_s calculations need to find all matchings containing a given node. This result can be cached. The necessary synchronization on the caches is handled by the Java standard library class `ConcurrentHashMap`.

The speed of the algorithm is mainly dependant on how often the function `SortByBoundedCost` in Algorithm 4 is executed. This means that the version of `match(...)` that pre-fixes some matchings can return quicker the more matching are fixed beforehand.

4.3.1 Approximating Optimal Matchings

The `match(...)` method is implemented using the customized Metropolis approach from Section 3.3. The pseudocode in Algorithm 4 shows a simplified version of the actual implementation. The main `Match` function first chooses an initial set of matchings (by invoking `Complete` with an empty set of fixed matchings). Then the Metropolis algorithm with the configured number of iterations is performed. Each consists of proposing a new set of matchings, calculating the acceptance probability for the new set and then accepting or rejecting it. Over all iterations the set of matchings with the lowest cost is saved and returned in the end.

To propose a new set, a random portion of the previous matchings is fixed and then completed to a set that satisfies the constraint that for every node in the left and right tree there is exactly one matching containing that node. Such a set can be passed to `AcceptanceProb` which uses `Cost` and the objective function described in Section 3.3 to determine the chance of accepting the matchings.

Crucial to the quality of the matchings that result from a run of the `match(...)` function is how `Complete` is implemented. The algorithm starts with the complete bipartite graph between the two trees. The matchings making up the graph are randomly shuffled to prevent any bias resulting from the iteration order. From this set all matchings are pruned that are not part of *fixed_matchings* but contain an artifact that is part of a matching in *fixed_matchings*. Then a loop is executed that runs until for each node in the trees only one matching remains that contains the node.

In each iteration of the loop, the bounded costs for all remaining matchings in the current subset of the bipartite graph are calculated and the matchings are sorted by their lower (and then upper) bound. Then a matching (that has not previously been fixed) is chosen from this list. To find a matching to fix, the authors of the original paper describe traversing the list in order of increasing bound and considering each matching with a fixed probability p. To avoid this iteration a probability distribution with probability mass function $P(X = k) = p \cdot (1 - p)^k$ is sampled using the inversion method until a valid index into the remaining matchings is found. This method ensures that less costly matchings are more likely to be fixed as they have a lower index in the sorted list.

4.3.2 Integration

Several options exist for integrating the `CostModelMatcher` into JDime. The most naive way of using it is to replace the previous matchers entirely as the `CostModelMatcher` should theoretically be able to deal with all situations that they covered. This however forces JDime to use Flexible Tree Matching even in situations that are not complicated enough to warrant the complexity of the Flexible Tree Matching algorithm. One such case is the sanity check of matching a tree with itself. In this case the `CostModelMatcher` has the same runtime as with any other pair of trees. This is unacceptable compared

Algorithm 4: The Approximation Algorithm for Optimal Matchings

```
Function Match($L$, $R$) is
    matchings ← Initialize($L$, $R$)
    lowest ← matchings
    for $i \leftarrow 0$; $i <$ number_iterations; $i \leftarrow i + 1$ do
        next_matchings ← Propose(matchings)
        if Chance(AcceptanceProb(matchings, next_matchings)) then
            matchings ← next_matchings
        end
        if Cost(next_matchings) < Cost(lowest) then
            lowest ← next_matchings
        end
    end
    return lowest
end
Function Propose($matchings$) is
    index ← RandomInt(Length($matchings$))
    fixed ← Sublist($matchings$, index)
    return Complete(fixed)
end
Function Complete($fixed\_matchings$) is
    current ← Bipartite($fixed\_matchings.L$, $fixed\_matchings.R$)
    fixed ← Copy($fixed\_matchings$)
    PruneAll(current, fixed)
    while Length(fixed) ≠ Length(current) do
        SortByBoundedCost(current)
        fixable ← RemoveAll(Copy(current), fixed)
        to_fix ← Get(fixable, SampleInt(Length(fixable)))
        Add(fixed, to_fix)
        Prune(current, to_fix)
    end
    return fixed
end
```

to the minimal runtime of the `EqualityMatcher` that dealt with this situation previously.

To minimize the amount of edges the `CostModelMatcher` has to consider, but still use it to improve the quality of matchings, two approaches were implemented. One is to use the `CostModelMatcher` as a post processing step over the matchings that were produced by the classical matchers. To that end the pre-fixing capability described in Section 4.3 is used. All classical matchings are fixed and the `CostModelMatcher` is used to try and match the previously unmatched nodes.

The second approach is to use the `CostModelMatcher` during the execution of the classical matchers (see Algorithm 2) whenever a subtree was not fully matched. A threshold was implemented to determine how little of the subtree has to be matched for the `CostModelMatcher` to become active. If using the `CostModelMatcher` leads to more nodes being matched, its matchings are returned instead of those of whatever concrete matcher was chosen.

These options are compared and evaluated in Chapter 5.

5 Evaluation

The success of the addition of the `CostModelMatcher` to JDime is evaluated using a set of test cases that contain situations in which JDime hopes to improve the merge results over unstructured tools. Different JDime configurations are used to produce sets of matchings for these scenarios. For each test case, an optimal set of reference matchings was constructed by hand if the classical matcher could not produce it.

To compare the sets of matchings against the reference, the Jaccard similarity coefficient is used. For two sets A and B it is defined as $J(A, B) = 1$ if both A and B are empty, otherwise it is

$$J(A, B) = \frac{|A \cap B|}{|A \cup B|}.$$

For every configuration, the similarity of the matchings to the reference, and the runtime of the configuration, are contrasted.

5.1 Setup

Four configurations of JDime (as introduced in Section 4.3.2) will be compared. The names defined in Table 5.1 are used to refer to the configurations in the rest of this chapter.

Table 5.1: The JDime Configurations

Name	Description
C_1	The JDime matchers without the addition of the `CostModelMatcher`. This provides the baseline this thesis hopes to improve upon.
C_2	The `CostModelMatcher` as a replacement of the classical matchers.
C_3	The `CostModelMatcher` as a post-processing step after the classical matchers.
C_4	The `CostModelMatcher` as an integrated component of the `Match` method from Algorithm 2.

5.1.1 Merge Tasks

Six merge tasks were constructed to test the effectiveness of Flexible Tree Matching in C_2 to C_4 compared to C_1. Most were deliberately chosen to exhibit weaknesses of the previous JDime matchers.

The `MovedMethod` task in Listing 5.1 shows the strength of structured merging. Traditional unstructured tools produce a conflict when faced with the reordering of methods in a class. In the AST, this change is represented by the reordering of the children of the nodes representing the class declaration. JDime can deal with this situation by performing unordered matching. This test case is included to determine whether the `CostModelMatcher` can handle reordering as well, specifically since the ordering cost c_o was added in such a way that it should not punish edges in this case.

```java
public class MovedMethod {

  private void foo() {
    System.out.println("Hallo");
  }

  public int bar() {
    return 42;
  }
}
```

```java
public class MovedMethod {

  public int bar() {
    return 42;
  }

  private void foo() {
    System.out.println("Hallo");
  }
}
```

Listing 5.1: Merge Task 1: MovedMethod

The first limitation described in Section 4.2.1 is exemplified by the test case in Listing 5.2. The renamed methods can not be matched by JDime since the method declaration nodes have different labels. Matching stops at the declaration nodes and the trees representing the bodies of the methods are never examined. Using Flexible Tree Matching, the hope is to match the method declaration nodes and all their children.

In Listing 5.3 a code fragment was surrounded with a for-loop construct. In the AST, this means adding several layers of nodes representing the loop before the subtree of the surrounded fragment occurs. A challenge here is that the loop, as well as the surrounded fragment contain a variable assignment producing a similar AST.

Listing 5.4 represents a similar problem as Listing 5.3 does, however with the added complexity of the `catch` block. The `CostModelMatcher` is expected to match the surrounded code but not the additional nodes of the surrounding construct.

```java
public class RenamedMethod {

  int getAnswer() {
    return 42;
  }
}
```

```java
public class RenamedMethod {

  int getResult() {
    return 42;
  }
}
```

Listing 5.2: Merge Task 2: RenamedMethod

```java
public class SurroundWithLoop {

  public boolean isOnline() {
    boolean online = false;

    online = check();

    return online;
  }

  public boolean check() {
    return true;
  }
}
```

```java
public class SurroundWithLoop {

  public boolean isOnline() {
    boolean online = false;

    for (int i = 0; i < 5; i++) {
      online = check();
    }

    return online;
  }

  public boolean check() {
    return true;
  }
}
```

Listing 5.3: Merge Task 3: SurroundWithLoop

```java
public class SurroundWithTry {

  public void doSmth() {
    String s = ex();
  }

  private String ex() {
    throw new RuntimeException();
  }
}
```

```java
public class SurroundWithTry {

  public void doSmth() {
    try {
      String s = ex();
    } catch (RuntimeException e) {
      e.printStackTrace();
    }
  }

  private String ex() {
    throw new RuntimeException();
  }
}
```

Listing 5.4: Merge Task 4: SurroundWithTry

In Listing 5.5 the left lines 7 and 8 are surrounded by another common construct, the `if` block. Matching the surrounded code fragment is complicated by the addition of the right line 12 whose AST has a similar structure and is on the same level as the one contributed by the left line 8.

```java
 1  import java.util.List;
 2
 3  public class ShiftedCode {
 4    public List<String> l;
 5
 6    public int firstLength() {
 7      String s = l.get(0);
 8      return s.length();
 9    }
10  }
```

```java
 1  import java.util.List;
 2
 3  public class ShiftedCode {
 4    public List<String> l;
 5
 6    public int firstLength() {
 7      if (l != null) {
 8        String s = l.get(0);
 9        return s.length();
10      }
11
12      return 0;
13    }
14  }
```

Listing 5.5: Merge Task 5: ShiftedCode

In addition to these five merge tasks a test case in which an AST is matched against itself was added to act as a sanity check. This case will be labeled as `Sanity`.

5.1.2 Parameters

To generate optimal matchings, the Flexible Tree Matching algorithm requires a domain specific configuration via its parameters. The weights w_n, w_r, w_a, w_s and w_o have a major influence on the matchings the algorithm produces. Other than that, the parameters that have to be fixed are the fixing probability p in `SampleInt` of Algorithm 3, the scaling factor β in the objective function of the Metropolis algorithm, and the number of iterations to run when approximating optimal matchings.

In Section 8 of their paper Kumar et al. describe their approach to learning a cost model that produces mappings with desirable characteristics for their use case. They set up a number of reference matchings and then use the generalized perceptron algorithm to learn weights that minimize the cost of these matchings. The weights that were used to generate the results in Section 5.2 were learned using a different technique.

Given the reference matchings described in the previous section, the Jaccard index provides a convenient way of calculating the quality of a set of matchings. A fitness function was defined as the average similarity (over all test cases) of the results of the standalone `CostModelMatcher` and the reference matchings. This fitness function takes as an input the weights to use for producing matchings. For a single chromosome of five values from the range $[0, 1]$ corresponding to the weights, the fitness function was maximised using an evolutionary algorithm provided by the Jenetics[1] library. After 100 generations, the cost model $w_r = 0.7$, $w_n = 0.9$, $w_a = 0.45$, $w_s = 0.30$, and $w_o = 0.95$ was chosen. Increasing the generation count did not result in a better average fitness.

As in the original paper, p was set to 0.7. Recall that, when completing a subset of the complete bipartite graph to a set of matchings in which every node is covered by exactly one element, the probability of fixing the matching at index k is $P(X = k) = p \cdot (1 - p)^k$. Choosing p as 0.7 leads to matchings being chosen that have very low bounds, as these make up the lower indices of the available matchings.

The parameter β was fixed at 30 since all involved trees have a similar size. This choice leads to the probability of accepting a significantly higher cost set of matchings (for the merge tasks being considered) being under 40%. Kumar et al. do not mention how they arrived their choice for β, only that it is determined based on the size of the trees. An AST specific function for calculating β may be a topic of future work.

The Metropolis algorithm was configured to perform 200 iterations. For their use case, Kumar et al. used 100 iterations. For the merge tasks being evaluated, an increase in matching quality was observed by using 200 (but not more) iterations.

5.2 Results

The results were gathered on a machine equipped with an Intel Core i7-4790 clocked at 3.60 GHz with 16 GiB of RAM available. Table 5.2 shows the runtimes of the different configurations for the test cases. As the application is running multithreaded inside the Java Virtual Machine, the runtimes were averaged over 20 runs.

Table 5.3 contains the similarity of the matchings that the configurations produced to the reference matchings. The similarity is calculated by taking the aforementioned Jaccard coefficient.

[1] URL: `http://jenetics.io/` (visited on 15/09/2016).

Table 5.2: The Runtime Results in Milliseconds

#	Merge Task	C_1	C_2	C_3	C_4
1	MovedMethod	1.15	5,191.10	0.50	301.60
2	RenamedMethod	0.15	1,119.50	232.95	369.95
3	SurroundWithLoop	0.40	14,111.05	542.10	457.75
4	SurroundWithTry	0.30	11,900.75	600.15	1,153.60
5	ShiftedCode	0.25	12,909.45	699.90	2,657.85
6	Sanity	0.25	12,056.30	0.25	0.20

Table 5.3: The Similarity Results

#	Merge Task	C_1	C_2	C_3	C_4
1	MovedMethod	100.00%	80.85%	100.00%	100.00%
2	RenamedMethod	47.83%	100.00%	100.00%	100.00%
3	SurroundWithLoop	90.20%	92.45%	100.00%	90.20%
4	SurroundWithTry	78.72%	91.84%	95.83%	88.00%
5	ShiftedCode	58.82%	84.91%	92.16%	96.00%
6	Sanity	100.00%	84.75%	100.00%	100.00%

5.3 Discussion

The aim of this thesis was to evaluate whether Flexible Tree Matching could be applied to abstract syntax trees and improve the tree matching capabilities of JDime. With the data collected in Tables 5.2 and 5.3, this section will evaluate the increase in matching performance over the classical matchers and examine where the various configurations of the `CostModelMatcher` failed to produce optimal matchings. See Table 5.1 for a detailed description of the configurations.

5.3.1 Configuration C_1

The first column of Table 5.3 shows, as expected, that the old JDime matchers do not match the test cases, except `MovedMethod` and the sanity check, properly. In case 2, the reference matches the whole left AST to its right counterpart. C_1 only manages to match the nodes above the method declaration. For cases 3 to 5 the old matchers also match the upper levels of the ASTs but can not match the surrounded code pieces lower in the tree. C_1 still achives relatively high scores since the surrounded fragments only account for a small piece of the AST. The sanity test of matching a tree against itself is of course passed by the old matchers.

Even though C_1 does not come close to the reference matchings for cases 2 to 5, it fails fast. The early exit when two nodes do not match in their label leads to very low runtimes. Case 6 is matched by the `EqualityMatcher` which produces the expected matching after one traversal of both trees simultaneously. Only case 1 has a slightly longer runtime as an unordered match on the level of the method declarations must be performed.

5.3.2 Configuration C_2

In C_2 the `CostModelMatcher` is used as a replacement of the classical matchers. This leads to the expectedly large increase in runtime. Without the restrictions leading to an early exit in the old matchers, the runtime of the `CostModelMatcher` is mostly determined by the size of the trees and the number of iterations performed. As the latter stays constant at 200, the C_2 column of Table 5.2 reflects the size of the ASTs representing the test cases. Notably the `Sanity` case, representing the easiest possible matching scenario, takes as much time as the similarly sized cases 3 to 5.

With the increased runtime also comes an increase in the quality of the matchings. The difficult cases 2 to 5 are all handled better than in C_1. The `CostModelMatcher` even produces perfect matchings for the `RenamedMethod` merge task. The fact that cases 1 and 6 are not matched perfectly exposes a fundamental difficulty in matching ExtendJ ASTs. They contain a lot of small and similar subtrees. An empty list of method parameters produces the same structure as an empty list of visibility modifies on a field. When multiple of these small subtrees occur in the trees being matched, the `CostModelMatcher` tends to confuse them as they have the same labels and structure. Often the ancestry and sibling weights that lead to good matches in a situation like case 2 allow mismatching these small subtrees.

5.3.3 Configuration C_3

Configuration C_3 suffers a lot less from this problem. All cases except 4 and 5 are handled perfectly and the two suboptimal cases are matched better than with C_2. In case 2, one of the mentioned empty lists is mismatched. Case 5 is more interesting. In this case, the old matchers that run before the `CostModelMatcher` match the `return` statement in line 12 of the right side with the one in line 8 of the left side. This matching is fixed and the `CostModelMatcher` is unable to veto the decision even though matching lines 8 and 9 of the left and right sides would produce a lower cost matching.

In addition to the quality of the matchings, the runtime is also improved. As mentioned, the less nodes need to be considered, the faster the algorithm runs. As the old matchers match a large portion of the AST, the post processing `CostModelMatcher` runs a lot faster. A major upside is that in cases where the old matchers suffice (cases 1 and 6), the `CostModelMatcher` adds no additional runtime.

5.3.4 Configuration C_4

Configuration C_4 represents an attempt at remedying the vetoing problem mentioned in Section 5.3.3. The `CostModelMatcher` is used to try and improve the matchings returned by one of the older matchers if they are below a certain quality threshold. In the current implementation, this threshold is fixed, causing case 5 to be improved while cases 3 and 4 are not improved as much over C_1 as they are in C_3. In case 5, the erroneous matching of the `return` statement is remedied using C_4.

The runtime here depends on the threshold. If it is chosen to high, the recursive structure of the `Match` algorithm in combination with the classical matchers may lead to many calls to the `CostModelMatcher`. With the threshold chosen for C_4, the runtime stayed within acceptable limits.

5.3.5 Summary

Without further optimization, the `CostModelMatcher` should not be used as a standalone matcher for ASTs. Configuration C_2 shows an increase in runtime that is to large to make it useful. The strengths of the `CostModelMatcher` lie in matching difficult subtrees in isolation after the obvious matchings were produced by the classical matchers.

Barring a better selection criterion than a static threshold, configuration C_3 should be used. Future work may improve upon the integration of the `CostModelMatcher` into the `Match` algorithm, as now the results and runtime of C_4 are too unpredictable to be used. If C_4 can be better integrated, it may become preferable over C_3 as it solves the vetoing problem inherent in it.

6 Conclusions and Future Work

In this thesis a working implementation of Flexible Tree Matching was added to the structured merge tool JDime. To successfully apply the algorithm to abstract syntax trees, a new cost term was introduced and the weights of the cost model were adjusted. By learning appropriate weight factors and weighing matchings of certain types differently, the matching of previously difficult scenarios was improved.

However, applying the Flexible Tree Matching algorithm incurred significant performance costs. These costs were partially mitigated using caching and parallelization techniques. One focus of future work may be to further optimize the matching process. Not only in terms of making the `CostModelMatcher` faster but also by applying it more efficiently (e.g. by pre-fixing a greater number of matchings).

Furthermore, while the weights w_n, w_r, w_a, and w_s were tuned, other parameters of the Flexible Tree Matching algorithm remain. The number of iterations is a significant factor in the performance of the algorithm. Determining what the minimal value is that still produces acceptable matchings would be a valuable contribution. To that end other proposition strategies may be evaluated. One could try and use the exact cost of the edges from the last iteration to select a better set than the first j for the next iteration. In addition to that, the parameter p that determines the probability of fixing an edge in Algorithm 4 (by way of `SampleInt`) may also have an influence on the required number of iterations. Another open question is a way of determining an appropriate value for β in the objective function of the Metropolis algorithm.

The pre-fixing of edges between the trees being matched had a significant, positive effect on the runtime of the matcher. There were however situations in which the pre-fixing of edges had a negative effect on the quality of the matchings. A way of vetoing pre-fixed edges when significantly better alternatives exist while retaining the runtime advantages would also be a valuable contribution to the `CostModelMatcher`.

Bibliography

[ALL12] Sven Apel, Olaf Leßenich and Christian Lengauer. ‘Structured Merge with Auto-Tuning: Balancing Precision and Performance’. In: *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2012, pp. 120–129 (cit. on pp. 20, 21).

[Ape+11] Sven Apel et al. ‘Semistructured Merge: Rethinking Merge in Revision Control Systems’. In: *Proceedings of the 19th ACM SIGSOFT Symposium and the 13th European Conference on Foundations of Software Engineering.* ACM, 2011, pp. 190–200 (cit. on p. 21).

[Bil05] Philip Bille. ‘A Survey on Tree Edit Distance and Related Problems’. In: *Theoretical Computer Science* 337.1 (2005), pp. 217–239 (cit. on p. 9).

[CG95] Siddhartha Chib and Edward Greenberg. ‘Understanding the Metropolis-Hastings Algorithm’. In: *The American Statistician* 49.4 (1995), pp. 327–335 (cit. on pp. 13, 26).

[Cha+96] Sudarshan S. Chawathe et al. ‘Change Detection in Hierarchically Structured Information’. In: *Proceedings of the 1996 ACM SIGMOD International Conference on Management of Data.* ACM, 1996, pp. 493–504 (cit. on p. 6).

[DP16] Georg Dotzler and Michael Philippsen. ‘Move-Optimized Source Code Tree Differencing’. In: *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 2016, pp. 660–671 (cit. on p. 8).

[Flu+07] Beat Fluri et al. ‘Change Distilling: Tree Differencing for Fine-Grained Source Code Change Extraction’. In: *IEEE Transactions on Software Engineering* 33.11 (2007), pp. 725–743 (cit. on p. 5).

[For96] Scott Fortin. *The Graph Isomorphism Problem.* Tech. rep. 96–20. University of Alberta, Edmonton, Alberta, Canada, 1996 (cit. on p. 4).

[Gam+96] Erich Gamma et al. In: *Entwurfsmuster*. 5th ed. Addison-Wesley, 1996, p. 373 (cit. on p. 20).

[GJ75] Michael Garey and David Johnson. 'Complexity Results for Multiprocessor Scheduling under Resource Constraints'. In: *SIAM Journal on Computing* 4.4 (1975), pp. 397–411 (cit. on p. 9).

[Kuh10] Harold W. Kuhn. 'The Hungarian Method for the Assignment Problem'. In: *50 Years of Integer Programming 1958-2008: From the Early Years to the State-of-the-Art*. Springer Berlin Heidelberg, 2010, pp. 29–47 (cit. on p. 25).

[Kum+11a] Ranjitha Kumar et al. 'Bricolage: Example-Based Retargeting for Web Design'. In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 2011, pp. 2197–2206 (cit. on p. 14).

[Kum+11b] Ranjitha Kumar et al. 'Flexible Tree Matching'. In: *Proceedings of the 22nd International Joint Conference on Artificial Intelligence - Volume Three*. AAAI Press, 2011, pp. 2674–2679 (cit. on pp. 9, 14, 26).

[Leß12] Olaf Leßenich. 'Adjustable Syntactic Merge of Java Programs'. MA thesis. Department of Informatics and Mathematics, University of Passau, 2012 (cit. on pp. 1, 5, 18).

[LV04] Antoni Lozano and Gabriel Valiente. 'On the Maximum Common Embedded Subtree Problem for Ordered Trees'. In: *String Algorithmics* (2004), pp. 155–170 (cit. on p. 7).

[Men02] Tom Mens. 'A State-Of-The-Art Survey on Software Merging'. In: *IEEE Transactions on Software Engineering* 28.5 (2002), pp. 449–462 (cit. on p. 1).

[Yan91] Wuu Yang. 'Identifying Syntactic Differences Between Two Programs'. In: *Software: Practice and Experience* 21.7 (1991), pp. 739–755 (cit. on p. 25).

[ZSS92] Kaizhong Zhang, Rick Statman and Dennis Shasha. 'On the Editing Distance Between Unordered Labeled Trees'. In: *Information Processing Letters* 42.3 (1992), pp. 133–139 (cit. on p. 25).

Eidesstattliche Erklärung:

Hiermit versichere ich an Eides statt, dass ich diese Masterarbeit selbständig und ohne Benutzung anderer als der angegebenen Quellen und Hilfsmittel angefertigt habe und dass alle Ausführungen, die wörtlich oder sinngemäß übernommen wurden, als solche gekennzeichnet sind, sowie dass ich die Masterarbeit in gleicher oder ähnlicher Form noch keiner anderen Prüfungsbehörde vorgelegt habe.

Passau, 29.09.2016 ....

(Unterschrift)