A Bayesian Approach to Diagram Matching with Application to Architectural Models

David Mandelin
UC Berkeley
mandelin@cs.berkeley.edu

Doug Kimelman
IBM Corporation
dnk@us.ibm.com

Daniel Yellin
IBM Corporation
dmy@us.ibm.com

ABSTRACT
IT system architectures, as well as other systems, are often described by formal models or informal diagrams. In practice, there are often a number of versions of a model, e.g. for different views of a system, divergent variants, or a series of revisions. Understanding how versions of a model correspond or differ is crucial, yet little work has been done on automated assistance for matching models and diagrams.

We have designed a framework based on Bayesian methods for finding these correspondences automatically. We represent models and diagrams as graphs whose nodes have attributes such as name, type, connections, and containment relations, and we have developed probabilistic models for rating the quality of candidate correspondences based on various features of the nodes in the graphs. Given the probabilistic models, we can find high quality correspondences using search algorithms. Preliminary experiments focusing on architectural models suggest that the technique is promising.

Categories and Subject Descriptors
D.2.11 [Software Engineering]: Software Architectures;
D.2.2 [Software Engineering]: Design Tools and Techniques;
I.2.1 [Artificial Intelligence]: Applications and Expert Systems

General Terms
Algorithms,Design,Documentation

Keywords
architecture,models,diagrams,matching,Bayesian

1. INTRODUCTION
An essential part of quality in any engineering discipline is control of change. That dictates the need to review and understand changes prior to accepting them. In environments where models and diagrams are a primary design artifact, this means being able to compare a baseline diagram with a revised diagram to identify correspondences and discrepancies between them. Comparison is also necessary when integrating divergent variants of a model. Furthermore, finding correspondences can be an important part of recovering traceability information that was not recorded during original development because of deadline pressure.

While techniques have long existed for comparing textual artifacts, little work has been reported concerning comparison of the diagrams and models that are common in large-scale IT system development. In this paper we formulate the problem of finding a correspondence between a pair of diagrams (a mapping between elements of one diagram and elements of the other) and we introduce a Bayesian approach to solve the problem.

Examples of the kinds of diagrams and models to which this work applies, which are central to modern IT systems development processes, include: structured representations of requirements, system overviews, architectural specifications of systems, network topologies, object designs, state transition diagrams, and control and data flow representations of code. Although this work clearly applies more broadly, we have initially focused on architectural models, driven by the needs expressed by IBM architects.

1.1 Scenarios
Finding correspondences between diagrams and models has several applications in the system development lifecycle. Above, we discussed these primary scenarios:

- Reviewing and understanding revisions
- Understanding divergent variants for integration
- Contributing to recovery of traceability information

Additional scenarios include:

Maintaining consistency between views. In model-driven development, different aspects of the system are represented by different submodels. For example, one model may represent a functional view of the system, omitting non-functional requirements such as performance, while another

1This work addresses both models (detailed representations, with formal semantics) and diagrams in general (broader, less rigid representations, not necessarily having an underlying formal model). Henceforth, without loss of generality, we often refer simply to diagrams.
may represent an operational view, including non-functional requirements. Alternatively, one diagram may show elements of the system pertaining to security, while another diagram may show elements pertaining to high availability.

The different models may be created at different times and by different authors, which can make them inconsistent. Also, sometimes a designer changes one model but neglects to change the others; this is a problem for model-driven development, where diagrams must be consistent. Maintaining or restoring consistency requires an accurate understanding of the correspondences between diagrams.

Maintaining consistency between overlapping views of a system is a vexing problem in the daily work of IT architecture practitioners. Tool support for maintaining consistency will have a significant impact on productivity and quality.

**Identifying use of reference architectures.** Given a repository of architectural design documents, a user may want to find reference documents similar to her own design. Also, reference architectures change, so architects must be able to find and update customized uses of reference architectures within solutions. Finally, architects may need to find where a portion of a reference architecture has been used across a portfolio of solutions, or to determine which portions of a reference architecture have been used in a given solution. Comparing two diagrams for similarity is a critical sub-problem in solving this larger problem, and the large volume of data requires this to be done automatically.

All of these scenarios are drawn from the experience of the IBM community of IT architecture practitioners, and instances of each of these scenarios can be found in the case studies that form the basis for the experiments described later in the paper. In fact, the models employed in those experiments are excerpts of artifacts produced by IBM architects as part of customer engagements.

### 1.2 Contributions of this Paper

Today, determining correspondences between models is a tedious, error-prone, time-consuming manual process. Our goal is to achieve an automated means of determining the correspondences. This requires us to answer several questions: How do we represent models? Which features of models must be represented? What algorithms should be used to find correspondences? In this paper, we provide answers to these questions.

Note that although this problem seems similar to formal matching problems, such as the graph isomorphism problem, it differs in several important ways. Chiefly, there is often no exact solution to this problem, and even defining the “best” correspondence in theory is difficult. Instead, the problem here is to produce an intelligent “guess” as to the best match, mimicking, as much as possible, what a human expert would do. Second, the best matching is often not 1–1. Finally, human experts rely on many considerations in determining correspondences, including model element labeling, connectivity, spatial position, and commonly applied transformations.

The main contributions of this paper are:

- We have identified and described the model correspondence problem, an important problem in software engineering (Section 3).
- We have designed a framework that can support and combine many kinds of reasoning used in finding model correspondences in a disciplined, quantitative fashion (Sections 4 and 5).
- We have implemented the matching algorithm in a prototype and demonstrated its effectiveness on real system models (Sections 6 and 7).

Sections 8 and 9 present related and future work. Section 10 is our conclusion, and Appendix A presents NP-Hardness results.

### 2. DIAGRAM FEATURES

We focused on the problem of finding correspondences in the domain of IT operational modeling [15]. Operational modeling is used by architects in IBM Global Services as part of a development methodology for customized IT solutions, and is supported by an experimental tool, The Architects’ Workbench [1]. An operational model exists at several levels, from an application view of an IT solution to a physical topology laying out the hardware and software components and their relationships. Discussions with several lead IBM architects indicated that indeed, finding corresponding elements in operational models automatically would be beneficial in everyday practice.

The main features of an operational diagram can be abstracted to elements found in many other kinds of diagrams:

- **Labeled nodes.** A node represents a system component. Each node has text associated with it, usually indicating its purpose. A node may have other attributes, depending on the diagram type. For example, in an operational diagram, an attribute may indicate whether the node is internal or external to the solution. Another common attribute is type, such as “hardware component” or “software component”. Attributes may be shown in the diagram textually or pictorially.
- **Edges.** An edge represents a relationship or association. An edge can be labelled, and it links two or more nodes. In operational models, edges indicate communication paths that connect nodes.
- **Containers.** A container is simply a node that contains other nodes. In operational diagrams, for example, a server may contain multiple software components.
- **Groups.** Often nodes are grouped together semantically. For instance, in operational models, servers located in the same building may be grouped within a common region. (Groups are different from containers: a node may belong to multiple groups but can be in only one container.) Like nodes, groups have labels and relationships. For example, regions have an adjacency relationship that indicates a connection.

The information represented by system diagrams can be broadly classified into three types: (i) syntactic information (e.g., nodes, labels, containment, edges), (ii) semantic information (e.g., types, defined semantic attributes), and (iii) visual information (e.g., position, shape, and color of diagram elements). Incorporating all of these kinds of information is one of the major challenges of diagram matching.
3. MODEL CORRESPONDENCE PROBLEM

As introduced above, the model correspondence problem is the problem of finding the “best” correspondence between the elements of two diagrams. The first issue is how to define “best”. It may seem appealing to define “best” as the correspondence that preserves a specific semantic relationship between the two diagrams, but this definition would be difficult to apply in practice for several reasons.

First, there are many possible semantic relationships between diagrams and it is hard to know which applies. For example, in one case we may have a diagram pair \((D, D')\) where \(D'\) is a revision of \(D\), with the semantic relation “is a revision of”. In another case \(D\) may be a conceptual description of a system and \(D'\) a physical description, with the semantic relation “implements”.

Second, if the semantic relationship is known, defining it in precise detail would be difficult, and even a precise definition may have insufficient information to find the best correspondence. Also, we conjecture that generic matching techniques can go a long way in finding correspondences between diagrams without having to incorporate domain-specific knowledge.

Third, many diagrams found in practice have no formal semantics: they use informal notions of “boxes” and “lines” to convey context-specific architectural notions.

3.1 Human Reasoning about Correspondences

Despite the problems with formally defining the best correspondence, after careful examination, human experts can often identify good correspondences. Thus, we approached the problem by trying to mimic expert reasoning. We did this by manually finding the best correspondences for some diagram pairs and recording the reasoning principles we used to find the correspondences. We showed our findings to practicing architects, who confirmed that the example diagram pairs were of interest and that we reached our conclusions via sensible principles.

From our test we identified the following principles of reasoning about diagram pair correspondences:

- Most decisions are made using evidence about which nodes from one diagram of a pair correspond with which nodes from the other diagram.
- Evidence takes the form of having similar or dissimilar features. For example, if two nodes have the same label, that is strong evidence that they correspond. As another example, if two nodes are of different semantic type, that is evidence that they do not correspond.
- Every feature in the diagram can be important evidence, including text, connection and containment relationships, and geometric and pictorial attributes.
- Most of the correspondence can be filled in by identifying 1–1 mappings using evidence about node pairs.
- For a node pair \((n, n')\), sometimes there is some evidence that \(n\) and \(n'\) correspond, and other evidence that \(n\) and \(n'\) do not correspond. This requires a judgment call about which evidence is more important.
- Special kinds of evidence are required to find correspondences that are not 1–1. For example, if two nodes are both within the same container, this is evidence that they may be the products of splitting a node.

4. SOLUTION OVERVIEW

This section presents an overview of our solution, and it serves as a roadmap to Section 5, which gives a mathematical description of the algorithm.

We explain the main ideas of the algorithm by tracing its behavior on a toy diagram pair, shown in Figure 1. This diagram pair is highly simplified for presentation purposes, but it does exhibit some of the difficulties found in the production models, in particular rough similarities and correspondences that are not 1–1. Note that the tags (C1, C2, ...) are only for ease of reference and are not part of the node text labels.

**Feature Similarity.** The first step of the algorithm is to extract relevant node features and evaluate their similarity. For example, nodes with similar labels often correspond, so we define a feature consisting of the text label of a node. We also define a function that assigns a similarity value between zero and one for any pair of label strings. Given these definitions, we can compute label similarity for each potentially corresponding pair of nodes \((n, n') \in HLV \times LLV\). The similarity values can be interpreted as a “raw similarity score” for a particular feature for any node pair of the two diagrams. Section 6.1 describes the features we have implemented.

Table 1 shows the label similarity values for the example.
From Feature Scores to Probability. Note that a similarity value in itself does not indicate whether a pair of nodes correspond: it is unclear whether a similarity of 0.6 is low or high, or how to proceed if there are possible matches with equal similarity. We can answer these questions by transforming our raw scores to probabilities. Given a probability distribution on the similarity values, Bayesian inference will convert the similarity of \((n, n')\) into the probability that \((n, n')\) correspond. The results of inference are listed in Table 2, which clearly shows that based on labels, \((C3, P3)\) and \((C4, P4)\) are much higher probability than the alternatives, but that labels are inconclusive for the other nodes.

We call the combination of a feature definition, a similarity measure, and a probability distribution on similarity an evidencer, because it provides all the information needed to process one kind of evidence. Section 5.2 defines evidencers formally, and Section 6.2 explains how we train the probability distributions.

Multiple Evidencers. Table 2 illustrates a common scenario: one evidencer alone cannot find the best correspondence. Thus, our algorithm uses several evidencers. For example, note that \(C1\) should correspond to \(P1\): we know this in part because \(C1\) and \(P1\) are contained in corresponding regions (i.e., \(R1\) in HLV corresponds to \(R1\) in LLV). Our algorithm carries out such reasoning with a region-based evidencer. The region-based evidencer similarity function returns one of the values \{SAME, ADJACENT, FAR\}.

With two evidencers, we need a mechanism for weighing one kind of evidence against another. Our algorithm combines evidence using Bayesian inference on a joint probability distribution over all kinds of evidence. The results are shown in Table 3. Note that \(C1\) now matches to \(P1\) with probability ten times greater than any other node.

A key advantage of our technique is that it combines different kinds of data using a rigorous quantitative procedure. Another advantage is that the form of joint distribution we use, a naive Bayesian model, allows us to combine different kinds of evidence using only the results of training evidencers individually: there is no need to retrain when combining. Sections 5.4 and 5.5 treat this computation formally.

Correspondences that are not 1–1. Because HLV has 4 nodes and LLV 5, clearly there is no one-to-one correspondence. Splits (one-to-many mappings) and merges (many-to-one mappings) are common in practice. Experts identify splits and merges by combining several pieces of evidence: for example, connected pairs such as \(P2\) and \(P5\) are more likely to be products of a split, and nodes “left over” after matching up the rest are likely to be part of a split. Our probability model takes this into account by modeling the probability of splits explicitly based on the evidencers already mentioned as well as special evidencers for splits.

Maximizing Correspondence Probability. Now that we have the probabilities of 1–1 node and split/merge mappings, we can combine them into a probability for a correspondence between a diagram pair. Our model uses an unnormalized probability: the product of individual correspondence probabilities. Sections 5.3–5.5 describe the model formally. The correspondence probability provides a “score” for the match; the desired match is the maximum probability correspondence.

There are several techniques for finding the maximum probability correspondence. (See Section 5.6.) Our implementation uses a greedy search on the space of correspondences, adding a single node-to-node match at each step. In the example, greedy search finds the correspondence \((C1, P1), (C2, P2\) and \(P5), (C3, P3), (C4, C4)\), with probability \(0.075 \cdot 0.004 \cdot 0.957 \cdot 0.4258 = 0.00021\), which is in fact a reasonable correspondence.

### 5. BAYESIAN CORRESPONDENCE MODEL

#### 5.1 Set Correspondence

In order to describe the algorithm formally, we must first introduce notation for diagram correspondences. Let \(D\) and \(D'\) be diagrams whose nodes are sets \(N\) and \(N'\), respectively. Our core notion is the set correspondence, which equates sets of nodes in \(N\) with sets of nodes in \(N'\), allowing nodes to be left out. Formally, \(P\) is a partial partition of a set \(U\) iff \(P = \{p_1, p_2, \ldots\}\) where each \(p_i \subseteq U\) and \(p_i \cap p_j = \emptyset\) for all \(i \neq j\). A set correspondence is a tuple \(C = (S, S', f)\) where

\[
S \quad \text{is a partial partition of } N, \\
S' \quad \text{is a partial partition of } N', \\
f : S \rightarrow S' \text{ is 1–1}
\]

Set correspondences are slightly more general than are needed for practical diagram matching, so we will work with two restricted kinds of set correspondence:

- A **singular correspondence** is a set correspondence \(C = (S, S', f)\) where \(|s| = 1\) for all \(s \in S, S'\). This is iso-

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0.364</td>
<td>0.261</td>
<td>0.400</td>
<td>0.154</td>
</tr>
<tr>
<td>C2</td>
<td>0.080</td>
<td>0.231</td>
<td>0.303</td>
<td>0.345</td>
</tr>
<tr>
<td>C3</td>
<td>0.167</td>
<td>0.320</td>
<td>0.813</td>
<td>0.214</td>
</tr>
<tr>
<td>C4</td>
<td>0.105</td>
<td>0.100</td>
<td>0.148</td>
<td>0.609</td>
</tr>
</tbody>
</table>

Table 1: Label similarity values for running example.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0.033</td>
<td>0.024</td>
<td>0.029</td>
<td>0.025</td>
</tr>
<tr>
<td>C2</td>
<td>0.033</td>
<td>0.024</td>
<td>0.030</td>
<td>0.030</td>
</tr>
<tr>
<td>C3</td>
<td>0.025</td>
<td>0.028</td>
<td>0.902</td>
<td>0.024</td>
</tr>
<tr>
<td>C4</td>
<td>0.029</td>
<td>0.030</td>
<td>0.024</td>
<td>0.236</td>
</tr>
</tbody>
</table>

Table 2: Pairwise correspondence probabilities based on label similarity.

<table>
<thead>
<tr>
<th>P1</th>
<th>P2</th>
<th>P3</th>
<th>P4</th>
<th>P5</th>
</tr>
</thead>
<tbody>
<tr>
<td>C1</td>
<td>0.075</td>
<td>0.004</td>
<td>0.007</td>
<td>0.000</td>
</tr>
<tr>
<td>C2</td>
<td>0.006</td>
<td>0.055</td>
<td>0.078</td>
<td>0.061</td>
</tr>
<tr>
<td>C3</td>
<td>0.004</td>
<td>0.064</td>
<td>0.957</td>
<td>0.004</td>
</tr>
<tr>
<td>C4</td>
<td>0.000</td>
<td>0.005</td>
<td>0.004</td>
<td>0.426</td>
</tr>
</tbody>
</table>

Table 3: Pairwise correspondence probabilities based on both label and region similarity.

---

Our algorithm first finds correspondences for regions, so the results are available to the region-based evidencer to use in finding correspondences for nodes.

---

7Our algorithm first finds correspondences for regions, so the results are available to the region-based evidencer to use in finding correspondences for nodes.
morphic to a 1–1 mapping between nodes in \( N \) and \( N' \), allowing nodes to be left out.

- A split-merge correspondence is a set correspondence \( C = (S, S', f) \) such that if \( f(s) = s' \), then \( |s| = 1 \) or \( |s'| = 1 \). Split-merge correspondences allow one-to-many and many-to-one mappings, but not many-to-many mappings.

In the remainder of this section, we will use the notation \( ⟨s, s'⟩ \) to indicate that sets \( s \) and \( s' \) correspond, i.e., \( s' = f(s) \). We also overload this notation for nodes, so \( ⟨n, n'⟩ \) means that \( n' = f(n) \). Finally, we use \( \Delta n \) to indicate that \( n \) and \( n' \) do not correspond, i.e., \( n' \neq f(n) \).

Note that a set correspondence induces a correspondence on edges as well. There is a correspondence between edge \( a = (m, n) \) in \( D \) and \( a' = (m', n') \) in \( D' \) iff \( ⟨m, m'⟩ \) and \( ⟨n, n'⟩ \).

### 5.2 Evidencers

Informally, an evidencer comprises (a) a function that extracts a feature from a node, (b) a function that measures the similarity of features from two nodes, and (c) a probability distribution of similarity conditional on whether two nodes correspond. Together, these structures enable us to compute the probability that two nodes correspond based on one kind of evidence.

Formally, an evidencer is defined in terms of a random model. Let \( M \) and \( M' \) be random variables over the space of all possible models. Let \( n \) and \( n' \) be random variables over the set of nodes in \( M \) and \( M' \), respectively. Define an evidencer to be a function \( e_i \) of \( n, n', M, M' \) that returns a measure of the similarity of \( n \) and \( n' \). Because \( e_i(n, n', M, M'') \) is a function of random variables, it is itself a random variable.

For brevity, we will write \( e_i(n, n') \) for \( e_i(n, n', M, M') \). Given a collection of evidencers \( e_1, \ldots, e_k \), we will write \( e(n, n') \) for the tuple \( (e_1(n, n'), \ldots, e_k(n, n')) \), which represents all the evidence for \( (n, n') \). Finally, we will denote the matrix of \( e(n, n') \) for every \( (n, n') \) simply as \( e \), which represents all the evidence for the diagram pair.

As an example, the label evidencer is \( E_l = f_l(n, n') \) where \( f_l(n, n') \equiv \text{textsim}(\text{label}(n), \text{label}(n')) \). \text{textsim} is a function returning a numeric similarity for two strings, and \text{label} returns the text label of a node.

### 5.3 Correspondence Probability

In order to use evidence to find the best correspondence, we model the best correspondence as a random variable \( c \) that can take on the value of any set correspondence. Our estimate of the best correspondence is the one that has the highest probability given the evidence:

\[
\hat{c} = \arg \max_c P(c|e)
\]

We cannot write down the distribution \( P(c|e) \) directly because the sample spaces are too large and the distribution is too complex. Thus, we have developed a probability model based on certain conditional independence assumptions that breaks \( P(c|e) \) down into small, manipulatable parts.

For ease of understanding, we will first explain a simplified version of the model that applies only to 1–1 correspondences, which could be useful in some cases. Then, we will describe the complete model that assigns a probability to any split-merge correspondence.

### 5.4 Singular Correspondence Model

The singular correspondence probability model defines the probability of a singular correspondence conditional on the observed evidence. The model rests on several key assumptions, which keep the model and the required training and inference algorithms simple. It is not necessary for these assumptions to be true, but they must be good approximations in order for the matching algorithm to be accurate; this can be evaluated empirically.

The first assumption is that whether one pair of nodes corresponds is independent of whether any other pairs correspond, conditional on the evidence. Intuitively, this expresses the hope that the evidence contains all the information needed to find correspondences, and that no additional information about other correspondences is needed. This assumption, together with the fact that a singular correspondence may be viewed as a set of corresponding pairs, allows us to factor the probability as a product of pair correspondence probabilities. The notion of a corresponding pair must be extended to account for nodes that are not matched (abusing notation slightly to allow \( S \) to refer to the set of nodes in a partial partition):

\[
\text{pairs}(c) = \{(n, f(n))|n \in S\} \\
\cup\{(n, φ)|n \in N \setminus S\} \cup \{(φ, n')|n' \in N' \setminus S'\}
\]

Conditional independence allows us to define the probability as the product of the probability of the pairs:

\[
P(c|e) = \prod_{(n, n') \in \text{pairs}(c)} P((n, n')|e(n, n'))
\]

We will treat the one-to-none and one-to-one elements of \( \text{pairs}(c) \) separately.

**One-to-none.** We assume simply that a node maps to nothing with fixed probability \( P((n, φ)) = m_0 \). We chose the numerical value of \( m_0 \) based on the empirical frequency of one-to-none pairs. It may improve accuracy to develop a model of the probability that \( n \) maps to nothing based on the features of \( n \). However, we have not implemented such models.

**One-to-one.** For this case, we need to factor the model further in order to separate the evidencers in \( P((n, n')|e(n, n')) \). Training a model in which all evidencers are joined would require us to specify the set of evidencers ahead of time and train all evidencers together. Training all evidencers together is difficult because the resulting distribution has a larger sample space, so it requires much more data to train.

We can separate the evidencers using our second key assumption, which is that the evidence similarity values are independent conditioned on whether a pair matches. The resulting model is often referred to as a *naïve Bayesian model* or a *generative model*.

226
Applying Bayes’ Theorem, followed by our conditional independence assumption, decomposes the probability into a factor for each evidencer:

\[
P((n, n')|e(n, n')) = \frac{P((n, n'))P(e(n, n')|(n, n'))}{P(e(n, n'))}
\]

\[
= \frac{P((n, n')) \prod P(e_i(n, n')|(n, n'))}{P(e(n, n'))}
\]

Note that the numerator is a product of separate conditional probabilities for each evidencer. Each conditional probability measures the likelihood that \(n\) and \(n'\) have a given similarity on a particular feature assuming that \(n\) and \(n'\) correspond (or not).

The denominator is not factored, but fortunately we do not need to compute the denominator directly. Instead, we use a standard normalization trick:

\[
p(0) \equiv P((s, s')|\emptyset) \prod P(e_i(s, s')|\emptyset)
\]

\[
p(1) \equiv P((n, n')) \prod P(e_i(n, n')|(n, n'))
\]

\[
P((n, n')|e(n, n')) = \frac{p(1)}{p(0) + p(1)}
\]

In order to evaluate this formula for a model pair, we need the conditional distributions \(P(e_i(n, n')|(n, n'))\) and the prior, \(P((n, n'))\). The conditional distribution can be determined by estimation from training data, which we describe in Section 6.2. For the prior, we selected a uniform distribution:

\[
P((n, n')) = m_1/N'
\]

for all \((n, n')\), where \(m_1 = 1 - m_0\).

### 5.5 Split-Merge Correspondence Model

The split-merge correspondence probability model is like the singular correspondence model, except that pairs of sets play the role of pairs of nodes. Pairs may be one-to-one, one-to-many, or many-to-one. Thus, we decompose a split-merge correspondence \(c = (S, S', f)\) into pairs as follows:

\[
spairs(c) = \{\{(s, f(s))|s \in S\} \cup \{\{\phi, \{n'\}|n' \in N' \setminus S'\} \}
\]

Given this decomposition of \(c\), we make independence assumptions similar to those in the singular correspondence case, arriving at a formula for \(P(c|e)\):

\[
P(c|e) = \prod_{(s, s') \in spairs(c)} P((s, s')|e(s, s'))
\]

The only difference between the split-merge case and the singular case is that for split-merge correspondences, we require a distribution for computing the probability \(P((s, s')|e(s, s'))\) where \(s\) or \(s'\) may have more than one element.

As in the singular case, we use a naïve Bayesian model, which yields:

\[
P((s, s')|e(n, n')) = \frac{P((s, s') \prod P(e_i(s, s')|(s, s'))}{P(e(s, s'))}
\]

As before, we can normalize to avoid computing the denominator. For the numerator, we handle the one-to-one and one-to-one cases exactly as we did for the singular model, except that \(m_1 \neq 1 - m_0\) because there is also the possibility of a match to multiple nodes. We simply define \(m_1\) to be a constant, and as with \(m_0\), we chose the value based on the empirical frequency of one-to-one pairs. The one-to-many case is the major difficulty.

In a one-to-many match, either \(|s| = 1\) and \(|s'| > 1\) or \(|s| > 1\) and \(|s'| = 1\). Below, we will refer to the singleton set as the merged node, and the other set as the split nodes. For expository purposes, assume that \(s = \{n\}\) and \(s' = \{n_1', n_2\}\).

The model for one-to-many matches is similar to the one-to-one model, but there are two new problems. First, evidencers work on node pairs, but in this case we have a pair of a merged node with multiple split product nodes. Second, there are diagram features that can be used to infer splits that cannot be represented by node pair evidencers. For example, if \(n\) has label “Database Services”, \(n_1'\) has label “Primary Database Services”, and \(n_2'\) has label “Backup Database Services”, the common suffix of \(n_1'\) and \(n_2'\) is equal to the label of \(n\), which suggests that \(n_1'\) and \(n_2'\) are the products of splitting and refining \(n\).

We solve the first problem by taking the similarity of \(\{n\}\) and \(\{n_1', n_2'\}\) for a given node pair evidencer to be the minimum of the similarity of any node pair:

\[
e_i(\{n\}, \{n_1', n_2'\}) \equiv \min(e_i(n, n_1'), e_i(n, n_2'))
\]

We also need to define the distribution for the split-merge extension of \(e_i\). We could do this by training, but because our training sets had only a small number of splits, we simply used the distributions from the 1–1 case as an estimate.

We solve the second problem by defining additional evidencers that operate on the split and merged nodes. These evidencers check for the conditions (a) the names of all nodes match almost exactly (as in “Server 1, Server 2”), (b) the common suffix of the merged node label matches the split node labels (c) a concatenation of the split node labels matches the merged node label, (d) the split nodes are contained in the same node, and (e) the split nodes are all connected. These evidencers are simple to implement and can be trained even with a small training set because their similarity measure is a boolean value.

We define the prior for one-to-many matches similarly to the other priors. Our current implementation assumes that \(n\) maps to at least two nodes with fixed probability \(m_{2+} = 1 - (m_0 + m_1)\), and that all possible \(k\)-node targets have equal probability, yielding:

\[
P((\{n\}, \{n_1', \ldots, n_k\}) = \frac{m_{2+}}{\binom{k}{2}}
\]

### 5.6 The Maximization Problem

The previous sections showed how to compute \(P(c|e)\) for a given correspondence \(c\) and evidence \(e\). To complete the algorithm, we must describe how to find \(c\) with maximal \(P(c|e)\).

Although the the exponential number of possible correspondences makes exhaustive search infeasible, the problem can be solved quite nicely if we restrict ourselves to singular correspondences and simple evidencers. A simple evidencer (a) computes conditional scores for individual node pairs \(P(e_i(n, n')|(n, n'))\), (b) does not rely on an initial correspondence to perform this computation, and (c) runs in polynomial time (ideally constant time per node pair, quadratic overall). To find the maximum probability cor-
respondence, construct a graph which has as its nodes the union of the nodes in the two diagrams, \( N \cup N' \). Place an edge from every node \( n \) in \( N \) to every node \( n' \) in \( N' \) with edge weight \( w(n, n') = P((n, n') \mid \epsilon(n, n')) \). We can now find the maximum probability correspondence in polynomial time using maximum-weight bipartite matching \([13]\).

If we allow split-merge correspondences, the mapping is not \( 1 \rightarrow 1 \) on nodes, and bipartite matching does not apply. In fact, the problem is NP-hard (see Appendix A for proof).

If we violate (b), some evidencer expects an initial correspondence (which could be obtained by running the bipartite matching algorithm without that evidencer) but that evidencer’s output needs to be considered in constructing a new maximal correspondence (hence the bipartite matching must be run again, and this can continue). The local connectivity evidencer (see Section 6.1) is an example of an evidencer that requires an initial correspondence.

Obviously, we cannot run in polynomial time if we violate (c). One case where this arises is when we consider global properties of the correspondence (as opposed to properties relating to a single node pair or a few nodes). For example, we may want to maximize the number of matching edges induced by the correspondence. This is a known NP-complete problem, the Largest Common Subgraph Problem \([6]\).

For all of these reasons, we approximate the maximal solution using search algorithms. We formulate the search over a state space where each state represents a correspondence. The transitions represent the effect of adding one node pair to the correspondence.\(^3\) In this transition system, all states can be reached from the state representing the empty correspondence. The goal of search is to find a close approximation of the maximum likelihood correspondence without searching for too long.

The simplest search algorithm is greedy search. In greedy search, we keep track of only piece of information, the current state. On each step, we examine all states reachable by a single transition from the current state, and move to the state with greatest probability. If there is no next state with greater probability than the current state, the search stops.

Although greedy search is very simple, we found that it performed well for many diagram pairs (see Section 7.) We expected greedy search to perform well because the probability model assigns very high or very low probabilities to most node pairs, so adding one more node pair that has a high probability is usually the right decision for the global optimization problem.

We have observed that greedy search fails in some cases with splits and merges, and especially with the connection evidencer, because matching edges requires more lookahead. We believe these problems can be solved with a more intelligent search algorithm, such as A*, or with heuristics that provide additional lookahead to the greedy search. Applying these ideas is future work.

6. EVIDENCERS

6.1 Evidencer Similarity Measures

Label Similarity. This evidencer measures the similarity between text labels of a node pair. We used an off-the-shelf string similarity measure, the Python standard library function `difflib.SequenceMatcher.ratio()`, which returns the fraction of characters in both strings that are contained in common substrings.

This string similarity measure is not particularly adapted to model matching, but it was sufficient for our prototype. One issue is that it gives as much weight to a common, uninformative substring such as “service” as to a more informative substring such as “Apache”. Our implementation mitigates this problem by replacing a few common words such as “service” and “node” with short, distinct strings such as “!”.

Future implementations could incorporate measures that weight common substrings according to rarity.

Region Similarity. As mentioned in Section 2, diagrams may have regions that contain nodes. A region may have a name, a set of neighboring regions, and a set of contained nodes. Empirically, the regions in a diagram pair have a correspondence relationship, and corresponding nodes are usually contained in corresponding regions.

The region similarity measure assigns to a node pair a similarity \( r \in \{ \text{SAME, ADJACENT, FAR} \} \). \text{SAME} means the nodes are in corresponding regions. \text{ADJACENT} means that the region of the left node corresponds to a region that is adjacent to the region of the right node. \text{FAR} covers all other cases.

Type similarity. Model nodes may be annotated with types that can be used in matching. For example, some diagrams have nodes typed as being internal or external to the system. Corresponding nodes typically have matching types.

The precise definition of this evidencer depends on the type system used for the diagrams. The diagrams we used for testing had a type hierarchy with a small number of types. We defined similarity values \( \{ \text{EQUAL, CLOSE, FAR} \} \). \text{EQUAL} means the two nodes have the same type. \text{CLOSE} means one type is an ancestor or sibling of the other in the type hierarchy. \text{FAR} covers all other cases.

A generalized notion of type similarity would be useful for pairs in which each diagram has a different set of types. For example, consider a diagram pair \((D, D')\) in which \(D'\) is produced from \(D\) by applying a simple transformation \(f\) followed by some manual editing. If the simple transformation has a rule “transforming a node of type \(t\) yields a node of type \(t'\)”, we can assist the algorithm by defining \(t\) and \(t'\) to be \text{EQUAL}.

Position Similarity. As noted in the Section 2, nodes have visual attributes, such as their position on the page. We observed that diagrams in a matching pair often have closely matching layouts, so that corresponding nodes are in the same or nearby positions on the page. Thus, if we plot the two diagrams on the same coordinate system, we expect the Euclidean distance between matching nodes to be small.

One idea would be to measure distance using the \((x, y)\) pixel positions recorded in our diagram format, but this would make it difficult to train a single distribution, because different diagrams have different scales. To solve this problem, we divide the distance by the maximum distance between any two nodes, resulting in a normalized distance between 0 and 1.
Local Connection Similarity. This evidencer measures the degree to which two nodes connect to corresponding nodes as the fraction of edges touching \( n \) that have corresponding edges (see end of Section 5.1) touching \( n' \), and vice versa. Because it depends on edge correspondence, this evidencer requires a candidate correspondence for all other nodes as input.

6.2 Training Evidencers

We define the probability distributions for evidencers by training them on examples for which the matching is done by hand. The input of training is a set of diagram pairs annotated with the correct correspondences. The output is an estimate of the probability distribution \( P(e_i(n, n') | (n, n')) \).

Some evidencers, such as region and type, return discrete similarity measures, so their distributions are discrete. Others, such as label, position, and connection, return continuous similarity measures and have continuous distributions. We used standard statistical techniques for both kinds, described next.

6.2.1 Discrete Evidencers

Estimating a discrete distribution is simple: set the probability of an event equal to the observed frequency. For example, if 81% of matching nodes have similarity \( \text{EQUAL} \), then estimate \( P(e_i(n, n') = \text{EQUAL} | (n, n')) = 0.81 \).

6.2.2 Continuous Evidencers

We followed a typical approach, that of selecting the form of the distribution ahead of time (e.g., the normal distribution), and then estimating a small set of parameters (e.g., mean and variance).

Our continuous evidencers have a finite range (the unit interval), so we need to select a distribution with finite range. Specifically, we chose a cutoff normal distribution, meaning that the minimum allowed value is 0 and the maximum 1. Formally, a cutoff normal random variable is \( N_c(\mu, \sigma^2) = \max(0, \min(1, N(\mu, \sigma^2))) \) where \( N(\mu, \sigma^2) \) is a normal distribution with mean \( \mu \) and variance \( \sigma^2 \).

The sample mean and variance do not provide correct estimates of the mean and variance of a cutoff distribution, so instead we used a linear regression technique. If a set of values from a normal distribution is ordered and then plotted against values from a standard normal with equal cumulative probability, linear regression yields an estimate of mean and variance. We performed this procedure on the values inside the cutoff range.

7. EXPERIMENTAL RESULTS

To evaluate our algorithm, we developed a prototype in Python and used it to find correspondences for architectural model pairs drawn from IBM IT customer engagements.

There were 30 architectural models, including excerpts of architecture overview diagrams, system context diagrams, component models, and operational models. Space limitations preclude showing these models in detail, but they are similar to IEEE 1471 conceptual diagrams, context views, analysis overall views, and deployment views, respectively, as shown in [7]. Our models, however, are based on a richer metamodel [15], are more complex, and are larger-scale. Typically, our models have dozens of nodes, with the largest having 50 nodes. The set of models included a reference architecture model, an initial operational model, and the operational model enhanced by incorporation of elements from the reference architecture.

Not only are the these models individually more complex than the toy models used in Section 4, the models used in our experiments also exhibit a high degree of variation: some are very weak in terms of name similarity, some have a significant number of split/merge refactorings, some have no position data, in one pair a number of sections of one diagram of the pair are missing, and in one pair the diagram has been significantly rearranged. This provides a robust test of how well the algorithm generalizes from training data.

We drew 21 model pairs from the set of 30 models. The model pairs covered all of the scenarios described earlier in the paper, including revisions, variants, derivations, and reference architectures. We ran a cross-validation test of our algorithm, as follows. We randomly assigned the diagrams to one of the three groups and ran a trial for each group. In each trial, one group is designated the testing set, and the other two form the training set. We train the evidencers on the training set, and then run the prototype on the testing set. Thus, the training set always contained 14 diagram pairs.

For each model pair, we evaluated 7 different algorithms for finding a correspondence: a non-Bayesian baseline algorithm that matches each node in the first model to the node in the second model with the most similar label, the Bayesian algorithm using only the label evidencer, four versions of the Bayesian algorithm with label and one other evidencer, and the Bayesian algorithm with all evidencers.

Our goal was to measure the accuracy of the algorithms in finding correspondences. For each run we counted node pair matches that were found correctly by the algorithm (correct matches, \( M \)), node pair matches that were not found by the algorithm (false negatives, \( N \)), and incorrect node pair matches found by the algorithm (false positives, \( F \)). From these values, we define the accuracy as \( M / (M + N) \) and false positive rate as \( F / (M + N) \).

Table 4 gives the overall results.

<table>
<thead>
<tr>
<th>Algorithm</th>
<th>% Accurate</th>
<th>% False Pos.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Baseline (non-Bayesian)</td>
<td>74</td>
<td>39</td>
</tr>
<tr>
<td>Label only</td>
<td>77</td>
<td>32</td>
</tr>
<tr>
<td>Label and Type</td>
<td>77</td>
<td>32</td>
</tr>
<tr>
<td>Label and Position</td>
<td>78</td>
<td>30</td>
</tr>
<tr>
<td>Label and Connection</td>
<td>80</td>
<td>23</td>
</tr>
<tr>
<td>Label and Region</td>
<td>83</td>
<td>35</td>
</tr>
<tr>
<td>Full (all evidencers)</td>
<td>87</td>
<td>19</td>
</tr>
</tbody>
</table>

Table 4: Experimental results: overall accuracy and false positive rate for different combinations of evidence.

These results confirm our primary hypotheses: the Bayesian approach is indeed effective—it was more accurate and had fewer false positives than the baseline non-Bayesian algorithm, even when using just the label evidencer alone. Furthermore, the results confirm our conjecture that it helps to combine multiple kinds of evidence—the accuracy of the Bayesian algorithm using the full combination of all evidencers had an accuracy of 87% with a false positive rate of 19%, significantly better than the baseline or any smaller combination of evidencers. There were was only one diagram pair in which the full combination of evidencers performed poorly compared to any other combination.
Examining the outcomes of individual runs indicates that in fact a generic configuration of evidencers can succeed across a broad spectrum of models. There is not, as yet, a need to use specific configurations of evidencers for particular classes of models. In fact, we believe that we can achieve even higher accuracy with our general model by improving the individual evidencers and adding a few new evidencers. These results clearly suggest the promise of our approach.

8. RELATED WORK

The work described in this paper relates to three streams of related research: traceability of software artifacts, schema matching, and domain-specific model matching. Traceability between software artifacts is the ability to navigate between related artifacts, such as requirements, architecture designs, models, code, test cases, and defect reports [10, 3]. At a finer level of granularity, it provides the ability to navigate between the elements included in different artifacts (e.g., between classes and methods in a UML diagram and classes and methods in code). Traceability is considered an important ingredient in improving the efficiency and quality of software production, and is used throughout the software development lifecycle. For example, it can be used by project managers to verify that each requirement has been satisfied, by architects to check the correct implementation of design concerns, and by maintainers to access the impact of a change. Although a complete traceability solution requires explicit tool and process support, automatic discovery of relationships, as described in this paper, can be an important element in augmenting these other techniques. Other work done in this realm includes the work by Alexander [2], which uses simple textual mappings to automatically find and instantiate links from use case references to use cases. Sherba et. al. [11] describes a framework called TraceM for automating the management of traceability relationships. It includes the notion of integrators that discover relationships between software artifacts, such as between specifications and UML diagrams, and between source code and discussion groups regarding the code. They also establish these artifact relationships using text matching techniques. As indicated in Section 7, text matching alone is often insufficient to develop highly accurate relationships between artifacts.

More sophisticated techniques for reconciling and finding correspondences between models have been developed in the domains of database and XML schemas. The goal of schema matching is primarily to make it easier to integrate and translate information [4] that is conceptually similar but represented by different schemas. Representative examples of work in this area include LSD [5], Cupid [9] and CMC [14]. A good overview of this problem domain and solution approaches can be found in [8]. The Cupid system [9] matches data models primarily using two sorts of information: lexical and structural, and computes a similarity score for any two nodes based upon this information. This work does not have a formal model of evidence or a formal method for combining evidence, but instead relies on heuristics.

The LSD system [5] takes a machine-learning approach, where each individual evidencer (which they call a learner), discovers relationship mappings based upon a training set of data. They use a technique called stacking to learn how to combine the correspondences produced for each simple evidencer. Hence LSD is similar to our work in that it provides a general mechanism for incorporating different evidencers and combining their results. However, as pointed out in [14], LSD has several drawbacks. It assumes a large corpus of training data (instance data for the schema), and it must relearn the classifications for each schema. While LSD learns concepts (e.g., “car type” matches “car make”), we are learning how effective a particular evidencer is in predicting a match by turning a similarity score into a probability distribution using Bayesian techniques. Another difference between LSD and our work is that LSD considers only 1–1 mappings. The CMC [14] system is similar to the LSD system in spirit, but introduces the notion of confidence in a particular evidencer for a particular problem instance, and uses that confidence to weigh the way the different evidence is combined. The notion of confidence is also potentially useful in our domain, and is something we would like to explore.

There is also research in the literature that focuses on domain-specific rules to produce matches between diagrams of very specific types. A good example of this is the Reconciliation system [12], which helps software designers find inconsistencies between object interaction diagrams, and gives them guidance on how to handle these inconsistencies. While aspects of their system are similar to ours, the bulk of their work focuses on the consistency rules for interaction diagrams, and they use the Dempster-Shafer theory of evidence to guarantee that these beliefs obey some fundamental properties, and to combine different beliefs stemming from different consistency rules. In contrast, we use training data and Bayesian reasoning to produce probability distributions, thereby giving a sound and empirical basis to the correspondences we produce.

9. FUTURE WORK

There are three major areas of future work: improving implementation performance, embedding the algorithm in a practical tool, and improving the Bayesian framework.

The time taken for each experimental run was a few minutes. At this time scale, the algorithm is already useful, but improved speed would make it much more practical. The Python prototype was created without considering performance. Reimplementation in a compiled language and reorganizing the code to avoid redundant computations is expected to bring the time taken down to a few seconds.

A powerful practical matching tool would require the ability to report “confidence” for each node match (see [14] for one approach), and to highlight discrepancies and navigate correspondences, a significant UI challenge.

Finally, there are several refinements that could be applied to the Bayesian algorithm. First, other ways of combining evidence can be tried, such as logistic regression. Second, improved distribution fitting techniques could improve evidencer accuracy. A final example of a refinement would be to learn a thesaurus to aid in matching (e.g., discovering that “network security” always maps to “firewall”, perhaps similarly to LSD [5]) or simply as an additional benefit of the algorithm.

10. CONCLUSION

We have identified and described the model correspondence problem, an important problem in software engineering. We have designed a framework that supports the reasoning needed to solve the model correspondence problem. We have implemented and tested a matching algorithm
based on our framework, finding that it achieved 87% matching accuracy on a set of test diagram pairs. We believe that this work holds great promise for the future.

Acknowledgements

We gratefully acknowledge discussions with V.T. Rajan, whose insights helped shape this work. We also thank Martha Kimelman, for her help on debugging, running many of the experiments, and comments on the paper. Finally, we thank Brian Fields and AJ Shankar for proofreading.

APPENDIX

A. COMPLEXITY (NP-HARDNESS)

In this section we show that the Split-Merge Correspondence Problem is NP-hard.

Let $D$ and $D'$ be diagrams with nodes $N$ and $N'$, respectively. Let a PairScore be a relationship over $2^N \times 2^{N'} \times \mathbb{R}$. If $s \subseteq N$, $s' \subseteq N'$, and $(s, s', r) \in \text{PairScore}$ then we say that correspondence of nodes in $s$ with those in $s'$ has score $r$. If $(s, s', r) \notin \text{PairScore}$, then this correspondence has score 0. We use the shorthand $r = \text{PairScore}(s, s')$ to represent $(s, s', r) \in \text{PairScore}$. For a split-merge correspondence (see Section 5), we assume that either $|s| = 1$ or $|s'| = 1$. Given a split-merge correspondence $C$, $\text{Score}(C) = \sum_{(s, s') \subseteq C} \text{PairScore}(s, s')$. Given diagrams $D$ and $D'$ and a relationship PairScore, the Split-Merge Correspondence Problem (SMCP) is to find the correspondence $C$ such that $\text{Score}(C)$ is maximal. (The product formulation given in Section 5.5 can be converted to an equivalent sum formulation by taking logarithms.)

Lemma A.1. The Split-Merge Correspondence Problem is NP-hard.

Proof: Let $A$, $B$, and $C$ be sets of size $m$. Let $W$ be a set of triples, where each triple is of the form $(a, b, c)$, $a \in A, b \in B, c \in C$. The 3DM problem is to determine whether or not there exists $W' \subseteq W$, $|W'| = m$, such that each element in $A$, $B$, and $C$ appears in exactly one triple in $W'$. A 3DM problem is represented by the tuple $(A, B, C, W)$. This problem is known to be NP-Complete [6].

Let $(A, B, C, W)$ be an instance of the 3DM problem. We construct an instance of SMCP $(D, D', \text{PairScore})$ as follows: For every $a \in A$, create a node $n_a$ in $D$; for every $b \in B$, create a node $n_b$ in $D$, and for every $c \in C$ create a node $n_c$ in $D'$. Hence there will be $2m$ nodes in $D$ and $m$ nodes in $D'$. For every $(a, b, c) \in W$, create a tuple $(\{n_a, n_b\}, \{n_c\}, 1) \in \text{PairScore}$, where $n_a$ ($n_b, n_c$) is the node in $D$ ($D'$) corresponding to $a$ ($b, c$, respectively). We claim that there exists a solution $W'$ to the 3DM iff there exists a correspondence $C$ for the SMCP problem such that $\text{Score}(C) \geq m$.

$\Rightarrow$ Let $W'$ be a solution to the 3DM problem. Let the correspondence $C$ be as follows: for each $(a, b, c) \in W'$, add the correspondence $(\{n_a, n_b\}, \{n_c\})$ to $C$, where $n_a$ ($n_b, n_c$) is the node in $D$ ($D'$) corresponding to $a$ ($b, c$, respectively). Since $W'$ is a solution to 3DM, each node will appear in only one tuple, and therefore the correspondence $C$ created will be a valid correspondence (a node will appear in at most one $(s, s') \in C$). $C$ will contain $m$ pairs $(s, s')$, and for each such pair $\text{PairScore}(s, s') = 1$, so $\text{Score}(C) = m$.

$\Leftarrow$ Say that $\text{Score}(C) \geq m$. Since each $(s, s') \in C$ contributes a score of at most 1, there must be at least $m$ such pairs in $C$. Since there are $m$ nodes in $D'$, and since $C$ is a valid correspondence, there are at most $m$ such pairs in $C$; hence $C$ contains exactly $m$ such pairs. Each one of these pairs must be of the form $(\{n_a, n_b\}, \{n_c\})$, where $n_a, n_b,$ and $n_c$ correspond to elements in $a \in A, b \in B,$ and $c \in C$ respectively. Furthermore, $(a, b, c) \in W$. For each such pair $(\{n_a, n_b\}, \{n_c\})$, create a tuple $(a, b, c) \in W'$, $|W'| = m$. Since $C$ is a valid correspondence, if $(a, b, c), (a', b', c') \in W'$ then $a \neq a', b \neq b'$, and $c \neq c'$. Hence $W'$ is a solution to the 3DM problem. $\Box$

B. REFERENCES


