Finding a Needle in the Haystack: A Technique for Ranking Matches between Components

Naiyana Tansalarak and Kajal Claypool

Department of Computer Science,
University of Massachusetts - Lowell
{ntansala,kajal}@cs.uml.edu
http://www.cs.uml.edu/dsl/index.html

Abstract. Searching and subsequently selecting reusable components from component repositories has become a key impediment for not only component-based development but also for achieving the overall usability of component development environments and the ultimate re-usability of the components themselves. Component matching, a fundamental aspect of the component search problem, has been a well-studied problem, resulting in many different matching techniques such as keyword, facet, signature and specification matching techniques. However, each matching technique individually applied for component search often yields a small or large number of (sometimes irrelevant) hits. In this paper, we propose a disciplined combination of the different matching techniques to provide a ranked set of highly qualified components from component repositories. Our work is based on a unique *Quality of Match* (QoM) metric that measures the overall "goodness" of the match between two given components. In particular, we provide *qualitative* and *quantitative* analysis to evaluate the QoM of two given components based on component information. Moreover, we present *QoMym*, a QoM-based hybrid match algorithm, that combines the strengths of different matching techniques and provides higher accuracy than existing matching techniques.

1 Introduction

Component-based software engineering has gained popularity over the past few years as the preferred mode for software construction, spurring the development of both commercial and freeware off-the-shelf components. However, while the wide availability of components is essential for the success of component-based software engineering, retrieving the *qualified* components[1] from the large number of available off-the-shelf components has rapidly become a key challenge for software developers. Today, developers are faced with a lack of search tools that can effectively aid the procurement of qualified components from one or more heterogeneous repositories, based on a given query component or a given set of requirements.

[1] The *qualified* component is a component that is determined to be fit for use in the context of (i) meeting the core application requirements; and (ii) inter-operating with respect to *component model*, *syntactic*, *semantic*, *design* and *platform* requirements [5, 20] of previously developed components that are deployed as part of the new system(the system under consideration).

Many techniques ranging from *keyword-based* to full-fledged *specification-based* heuristics have been proposed in the literature [7, 12, 8, 21, 22, 6, 4] to provide effective retrieval of qualified components during the discovery process. The keyword-based approach [7] is simple and flexible as users simply specify the query as a set of keywords representing the component requirements in which they are interested. This approach while simple is also prone to low accuracy resulting in either too many or too few hits, or in some cases even completely unrelated hits [15]. The faceted approach [12] classifies components based on predefined taxonomies. While this approach provides a better description of components than a pure keyword-based approach, users must be familiar with the classification scheme to effectively retrieve a needed component. Moreover, it is often hard to manage classification schemes when domain knowledge evolves and as a result the component falls into two or more categories [15]. Signature matching approaches [21] decide the match between two given components, the query and library components, based on the signatures of the methods in the two components. While signature matching uses intrinsic built-in information about the component, that is its *type information*, it often still returns irrelevant hits [22, 4]. For example, consider the methods `strcpy` and `strcat` in the standard `C` library. These methods have the same signature but encode different behaviors. The specification matching approach [22, 6], introduced to overcome the problem of signature matching, uses the method's pre- and post-conditions that capture the functionality of the method. While specification matching provides more accurate hits, it is too time-consuming to be practical as its implementation, often based on theorem proving techniques, is expensive [4]. Another drawback of the specification approach is the practical lack of pre- and post-conditions in component code. An approach using test cases [4] that captures the partial semantics of the required functionality via method interactions attempts to address these drawbacks. While this approach tends to improve the performance of the discovery process, defining precise test cases that represent the required functionality is often too hard to describe.

While these techniques take steps in the right direction, each approach individually is limited in the *quality* of the matches produced, resulting often in too many (sometimes irrelevant) matches or in some cases even no matches. In this paper, we now propose a novel matching technique – a disciplined application of different matching algorithms to all aspects of a component thereby exploiting the diversity of semantic and syntactic information inherent in a component. The goal of our matching technique is to provide a ranked set of highly qualified matches from component repositories based on a given query component. Our work is based on a unique *Quality of Match* (QoM) metric that measures the "goodness" of a match. We define a *match taxonomy* and a weight-based *match model* that qualitatively and quantitatively classify the match between a query and a library component. These are based on information ranging from the type hierarchy to the labels of properties and methods of the two components. The match taxonomy and the match model together form the basis of the $\underline{QoM}$-based *hybrid match* algorithm, *QoMym*. The *QoMym* algorithm uses the match taxonomy as a guide for the algorithm execution, and also utilizes the weight-based match model to calculate the QoM for each element of the two components. We present a set of preliminary experiments that show the benefits of *QoMym* over using individual matching techniques

and other combination techniques, and provide an empirical measure of the quality of *QoMym*.

Roadmap: The rest of paper is organized as follows. Section 2 describes XCM, our unifying component description model, that provides a descriptive superset of information for components conforming to various component models in the market. XCM is the abstract component description model for QoMym. Section 3 presents the QoM-based match taxonomy, while Section 4 defines the weight-based match model, a quantitative measure of QoM. Section 5 introduces the QoMym algorithm. Experimental evaluation of the QoMym algorithm is given in Section 6. We conclude in Section 7.

2 The XML-based Unifying Component Description Model

In the context of component matching, the diversity in component models [10, 13, 14, 11, 2] imposes a restriction that often limits component searching to the features specified for a single component model. To extend component matching to encompass a heterogeneous set of components, we now present an XML-based unifying component description model, **XCM**, that crosscuts the information of components conforming to these diverse component models [18].

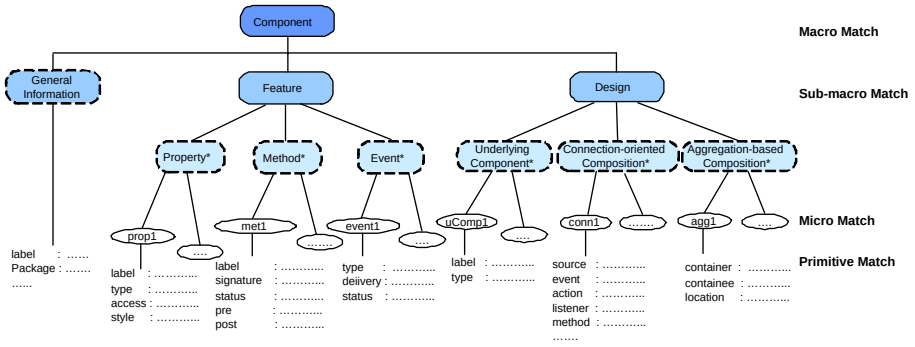

Fig. 1. The XCM Hierarchical Component Structure.

Figure 1 represents the XCM hierarchical component structure. Here, a component is defined via (i) `general information` that encapsulates the *label*, *version*, *package*, *language*, *component model*, *domain*, *operating system* and *publisher* of a component; (ii) `feature` that encapsulates the component's set of properties, methods and events; and (iii) finally `design`[2] that encapsulates the connection-oriented and

[2] The design is the description of the layout that describes how a composite component is constructed using a set of pre-existing components. A primitive component is a stand-alone component that does not rely on any other component for its functionality, while a composite component is constructed by either connecting (*connection-oriented*) or aggregating (*aggregation-based*) [1] a set of pre-existing components.

aggregation-based compositions of a set of pre-existing components. The hierarchical component structure in Figure 1 can be represented as an XML document, while the general structure of the description model – the XCM concepts – can be described as an XML Schema. Full details of the XCM component ontology can be found in [18].

3 The Qualitative Analysis - Match Taxonomy

In this section, we define a *match taxonomy* that qualitatively provides the quality of match (QoM) between two given components. The match taxonomy classifies the matches at the leaf, internal and root nodes of the XCM hierarchical structure (Figure 1) as (i) *primitive* match - match at the leaf level; (ii) *micro* match - match at the level of properties (or methods, events, underlying components, and connection-oriented as well as aggregation-based compositions of underlying components); (iii) *sub-macro* match - match at the level of features and design; and finally (iv) *macro* match - match at the level of the component. Each level of these matches is tightly coupled and hence heavily dependent on its lower level.

3.1 Primitive Match

A *primitive* match is the match between two corresponding leaf elements, that is the primitive information captured in the XCM structure, such as the label or the type (Figure 1). Each primitive element is classified as either *syntactic* or *semantic* based on the type of information encapsulated in the element. For example, a label imparts semantic information and hence is classified as a semantic element, while the domain type is regarded as a syntactic element.

The match between two given primitive elements is categorized as *exact* ($=$), *relaxed* ($\approx$) or *no match* ($\neq$). Dependent on the type of primitive elements, different matching techniques can be employed to determine the actual match and hence the classification between the two values of a given primitive element. For example, the label and the description of a property are best matched via a *linguistic* algorithm. A match is said to be *exact* in this case if the values are identical, *relaxed* if the values are synonyms, hypernyms, or come from the same stem, and a *no match* otherwise. Domain types, on the other hand, are best compared based on their relationships in the overall type hierarchy, requiring specialized type hierarchy comparison algorithms. Two domain types are said to be an *exact* match if their values are identical, *relaxed* if they are in the same path in the type hierarchy or *convertible* if there exists a known function to convert the source type to the target type, and a *no match* otherwise.

3.2 Micro Match

A *micro* match is the match between two given properties, methods, events, underlying components, connection-oriented or aggregation-based compositions of underlying components [3]. It is defined as a match of all its primitive elements.

[3] For the rest of the section, we describe the matches based on the properties. The match for the other features and design elements are similar.

The QoM of a micro match between two given properties is said to be (i) *exact* ($=$) - if all primitive elements of the source property match exactly to those of the target property; (ii) *relaxed* ($\approx$) - if either (a) all primitive elements of the source property have a combination of exact and relaxed matches in the target property, or (b) some (but not all) primitive elements of the source property have matches in the target property; or (iii) *no match* ($\neq$) - otherwise. Consider for example the two properties `String day` and `int day`. The micro match between these properties is *relaxed* as the labels are an exact match but the domain types are convertible.

3.3 Sub-Macro Match

Moving up the XCM hierarchy, we define a *sub-macro* match as the match between two sets of *component features* or two sets of *component design*. The match of each set is the collection of matches of its individual elements. The match of the component feature is thus defined on the basis of the matches of all properties, methods and events, while the match of the design is defined based on the matches of all underlying components and their compositions.

The QoM for a sub-macro match is classified based on (1) the number of micro matches; and (2) the quality of the micro matches. Based on the number of micro matches, the QoM of a sub-macro match is classified as either a *total* or a *partial* match. In a total match, all elements (properties, methods and events) of the source feature match some or all elements of the target feature, while in a partial match some (but not all) elements of the source feature match those in the target feature. Combining the two criteria, number of matches and the quality of micro matches, we now define four classifications for the QoM at the sub-macro level: *total exact*, *total relaxed*, *partial exact* and *partial relaxed*.

3.4 Macro Match

A match at the highest level, between two components, is termed a *macro* match. A *macro* match is defined as a match of the primitive elements of the two components, that is a match of their labels and descriptions, as well as a match of their non-primitive elements at the sub-macro level, that is their component features and design.

The QoM for a macro match is categorized on the basis of (1) the number and quality of the primitive element matches for labels, description and invariants; and (2) the number and quality of the sub-macro matches for the component features and design. Similar to the sub-macro match, the QoM at the macro level is classified as *total exact*, *total relaxed*, *partial exact* or *partial relaxed*.

4 The Quantitative Analysis - Weight-Based Match Model

Based on the qualitative analysis, it is often easy to evaluate when one match is better than the other. For example, an exact match is always better than a relaxed match. However, in some cases such a distinction between the QoM for two or more matches cannot be established as easily. For example, based on qualitative analysis alone we

cannot accurately determine whether a total relaxed match is better than a partial exact match, or one partial exact match is better than another partial exact or even a partial relaxed match. To address this deficiency, in this section, we now present a *weight-based match model* that quantitatively determines and ranks the QoM. We define this quantitative model at each level of the match taxonomy.

4.1 Primitive Element Match Model

Match between two primitive elements is classified as exact ($=$), relaxed ($\approx$) or no match ($\neq$). We term $=$, $\approx$ and $\neq$ the core *match operators* and assign a numeric weight to each of these match operators. The operator $=$ is assigned a weight of `1.0` to indicate an exact match, $\approx$ a weight ranging from `0.1` to `0.9` to denote a relaxed match, and $\neq$ a weight of `0.0` to represent a no match. These weights form the basis of the match model, and represent the *match weight* of two given primitive elements, denoted as $\mathcal{W}(\epsilon_s, \epsilon_t)$ where ϵ_s and ϵ_t are source and target primitive elements, respectively.

4.2 Micro Match Model

Based on the weight of the primitive matches, we now define the quantitative value for the micro match of two properties[4] as the normalized sum of the match weights of all its primitive matches. Formally, the QoM of a micro match, denoted as $QoM(\alpha_s, \alpha_t)$, is given as:

$$QoM(\alpha_s, \alpha_t) = \sum \mathcal{V}_\epsilon \; \mathcal{W}(\epsilon_s, \epsilon_t) \tag{1}$$

Here, (i) α_s and α_t are the source and target property; and (ii) $\epsilon_s \in \alpha_s$ and $\epsilon_t \in \alpha_t$ are the source and target primitive elements. Intuitively, it can be observed that not all primitive elements have an equal significance in determining the QoM between a source and a target properties. For example, for a property, the domain type typically has more significance than the property style. We thus specify $\mathcal{V}_\epsilon$ as the *significance value* of the specified primitive element ϵ. In keeping with this intuition, we assign significance values as *absolute numeric numbers* to all primitive elements of a property where the total significance value of all primitive elements for a property is `1.0`.

4.3 Sub-Macro Match Model

To provide a quantitative value for the sub-macro QoM, we define two measures, *micro match weight* and *cardinality ratio*. The micro match weight, denoted as $\mathcal{R}_W(\beta_s, \beta_t)$, is the normalized sum of QoM of all micro matches of the component feature (or design) and is given as:

$$\mathcal{R}_W(\beta_s, \beta_t) = \sum_{i=1}^{n} \mathcal{V}_i \frac{\sum QoM(\alpha_{si}, \alpha_{ti})}{|\beta_{si}|} \tag{2}$$

[4] The quantitative value for the micro match of two methods, events, underlying components or compositions of underlying components is defined in a similar manner.

Here (i) $\beta_s \in C_s$ and $\beta_t \in C_t$ are the source and target component feature; (ii) i is the element type that is the property, method or event type; (iii) n is the number of element types defined in the source component feature; (iv) $\alpha_{si} \in \beta_s$ and $\alpha_{ti} \in \beta_t$ are the source and target elements for the specified type i; (v) $|\beta_{si}|$ is the number of source elements for the specified type; (vi) $\mathcal{V}_i$ is the significance value of the specified element type i. For example, in the component feature, the method denoting the behavior of the component would typically have more significance than the property.

The cardinality ratio, denoted as $\mathcal{R}_S(\beta_s, \beta_t)$, is the ratio of the number of micro matches and the cardinality of the source component feature (or design) and is given as:

$$\mathcal{R}_S(\beta_s, \beta_t) = \sum_{i=1}^{n} \mathcal{V}_i \frac{|(\beta_{si})^m|}{|\beta_{si}|} \tag{3}$$

where $|(\beta_{si})^m|$ is the number of micro matches for the specified element type i.

The QoM of a sub-macro match, denoted as $QoM(\beta_s, \beta_t)$, is now defined as the normalized sum of the micro match weight and its cardinality ratio.

$$QoM(\beta_s, \beta_t) = \frac{\mathcal{R}_W(\beta_s, \beta_t) + \mathcal{R}_S(\beta_s, \beta_t)}{2} \tag{4}$$

4.4 Macro Match Model

The macro QoM, denoted as $QoM(C_s, C_t)$, is defined as the normalized sum of primitive element matches for label and description as well as the QoM of sub-macro matches. Formally, $QoM(C_s, C_t)$ is given as:

$$QoM(C_s, C_t) = \sum \mathcal{V}_\epsilon \mathcal{W}(\epsilon_s, \epsilon_t) + \sum \mathcal{V}_\beta QoM(\beta_s, \beta_t) \tag{5}$$

where (i) C_s and C_t present the source and target component; (ii) $\epsilon_s \in C_s$ and $\epsilon_t \in C_t$ are the source and target primitive element; (iii) $\mathcal{V}_\epsilon$ and $\mathcal{V}_\beta$ are the significance values of the specified primitive elements, and the component feature and design; and (iv) $\beta_s \in C_s$ and $\beta_t \in C_t$ represent the source and target component feature (or design).

5 The QoMym Algorithm

The *QoMym* algorithm is a depth-first match algorithm that is guided by the match taxonomy presented in Section 3 and is directly based on the match model given in Section 4. The overall execution of *QoMym* is depicted in Figure 2 and its pseudo-code is given in Figures 3 - 6. The *QoMym* algorithm first evaluates the match values for all primitive elements, that is all leaf nodes including the label, the description, the type etc. of the component itself. The primitive element matches are evaluated based on their type. For example, a linguistic algorithm is employed to determine the level of match between two labels (or descriptions). Our linguistic algorithm uses a combination of WordNet [9] and a domain-specific dictionary that includes commonly used abbreviations. A full description of the linguistic algorithm is beyond the scope of this paper. For more details please refer to [17]. Similarly, to match two domain types we have

developed an algorithm to compare the types along the specified type hierarchy. For better performance, the type hierarchies were converted using a dewey-based numbering scheme [19] that captured the relationship and the hierarchy of the given types. In particular, each primitive domain type of a component is automatically mapped into its dewey-based number before a match is determined.

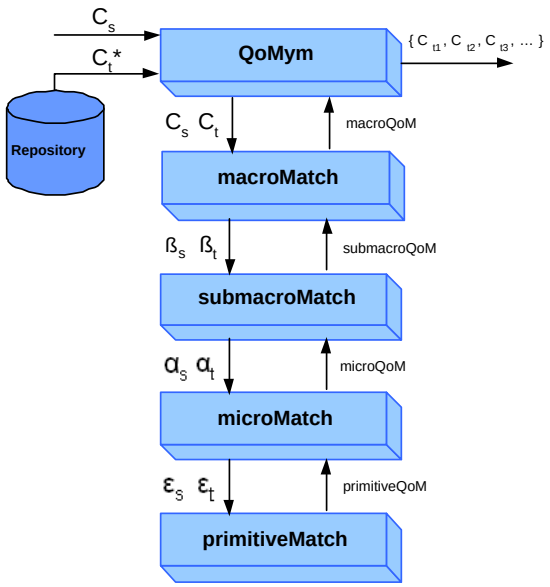

Fig. 2. The Overall Execution of QoMym.

Each source primitive element is compared with every target primitive element, and all match values above the threshold are saved. This threshold was determined empirically after running a set of controlled experiments (see details in Section 6). Once all primitive matches are computed, the micro matches are evaluated using the *microMatch* module given in Figure 4. All match values above a certain threshold are saved. The *submacroMatch* module given in Figure 5 determines the match value for the feature and the design. The submacro match values are used to determine a single macro match value - the match value of the source and target components - using the *macroMatch* module given in Figure 6. The QoMym module (Figure 3) finally returns a set of qualified components that have macro match values above threshold.

The running time for the algorithm lies in $\theta(|R||C_t||C_s|)$ where $|R|$ represents the number of components in the repository, and $|C_t|$ as well as $|C_s|$ the cardinality of a target and a source component.

```
Set QoMym (Component C_s) {

    result ← ∅;
    for each Component C_t ∈ Repository R {
        macroQoM = macroMatch(C_s, C_t);
        if macroQoM ≥ macroThreshold
            result ← result ∪ < C_t, macroQoM>;
    }

    return result;
}
```

Fig. 3. The QoMymAlgorithm.

```
double microMatch (Micro α_s, Micro α_t) {

    microQoM ← ∅;

    // ε_s ∈ α_s and ε_t ∈ α_t
    for each corresponding primitive pair (ε_s, ε_t) {
        primitiveQoM ← primitiveMatch (ε_s, ε_t);
        sig ← DB.getSignificance (ε_s.getType());
        microQoM ← microQoM + (sig * primitiveQoM);
    }

    if microQoM ≥ microThreshold
        return microQoM;
    else
        return 0;
}
```

Fig. 4. The microMatch Algorithm.

6 Preliminary Experimental Results

The goal of the *QoMym* algorithm is to improve the overall match quality of the qualified components retrieved in response to a specified query component. We conducted several experiments to evaluate the potential benefits of the *QoMym* algorithm over other existing algorithms. In this section, we describe our experimental setup and methodology together with our results.

6.1 Experimental Setup and Methodology

Figure 7 illustrates the overall architecture of the QoM system. The QoM system together with *QoMym* algorithm is implemented in Java (SDK 2.0) and deployed on a standalone PC Pentium IV 2.8 GHz with 512 Mb RAM running Microsoft Windows XP. The QoM system takes a query component as input, matches the query component against a set of library components using the *QoMym* algorithm, and returns the

```
double submacroMatch (Component C_s, Component C_t,
                      String submacroType) {

   // get source and target micro elements
   source ← C_s.getMicroElements (submacroType) ;
   target ← C_t.getMicroElements (submacroType) ;

   initialize microQoM[|C_s|][|C_t|];

   // calculate QoM for all (α_s,α_t);
   for each micro α_s ∈ source {
      for each micro α_t ∈ target {
         if α_s.getType() == α_t.getType()
            microQoM[α_s][α_t] ← microMatch (α_s, α_t);
   }

   // calculate the submacro QoM
   R_W ← getMicroMatchWeight (microQoM);
   R_S ← getCardinality (microQoM);
   submacroQoM ← (R_W + R_S) / 2;

   return submacroQoM;
}
```

Fig. 5. The submacroMatch Algorithm.

match results to the user. The repository used for the experiments contained JavaBean components across four domains: 24 GUI components, 12 data collection components, 8 calendar components, and 16 components for testing. These components were automatically introspected, transformed into XCM documents and subsequently loaded into the repository.

Measure Of Match Quality. To evaluate our approach, we compared the manually determined real matches (R) for a given match task[5] with the matches P returned by the match algorithm. We determined the true positives, i.e., the correctly identified matches, I; the false positives, i.e., the incorrectly identified matches, $F = P \setminus I$, and the false negatives, i.e., the missed matches, $M = R \setminus I$. Based on the cardinalities of these sets, the *Precision* and *Recall*[6] of the match algorithm were computed.

- $Precision = \frac{|I|}{|P|} = \frac{|I|}{|I|+|F|}$ estimates the reliability of the match predictions.
- $Recall = \frac{|I|}{|R|}$ specifies the share of real matches that is discovered by the algorithm.
- $Overall = 1 - \frac{|F|+|M|}{|R|} = \frac{|I|-|F|}{|R|} = Recall * (2 - \frac{1}{Precision})$ represents a combined measure of match quality, taking into account the post-match effort needed for both removing false matches and adding missed matches.

[5] Here a match task denotes the matching of a query component with the components in the repository to determine the qualified components.

[6] Precision and Recall are taken from Information Retrieval literature

```
double macroMatch (Component C_s, Component C_t) {

    // get the primitive QoM
    W_l ← linguisticMatch (C_s.getLabel(), C_t.getLabel());
    W_d ← lingusiticMatch (C_s.getDescription(), C_t.getDescription());

    // get the submacro QoM
    subQoM_f ← submacroMatch (C_s, C_t, "feature");
    subQoM_d ← submacroMatch (C_s, C_t, "design");

    // get the significant values
    V_f ← DB.getSignificance ("feature");
    V_d ← DB.getSignificance ("design");
    V_l ← DB.getSignificance ("label");
    V_d ← DB.getSignificance ("desc");

    // calculate the macro QoM
    macroQoM = (V_l * W_l) + (V_d * W_d) +
               (V_f * subQoM_f) + (V_d * subQoM_d);

    return macroQoM;
}
```

Fig. 6. The macroMatch Algorithm.

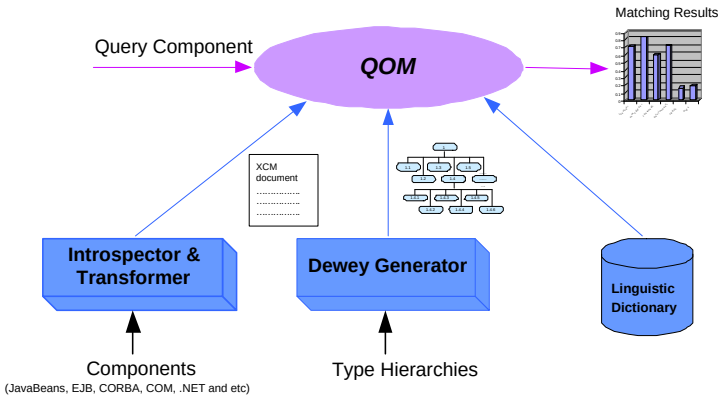

Fig. 7. The Overall Architecture of the QoM System.

6.2 Determining Threshold and Significance Values

The accuracy of the *QoMym* algorithm is dependent on the *threshold* and *significance values* that are an integral part of the weight-based match model defi ned in Section 4. To determine optimal values for these parameters (threshold and signifi cance), we conducted a set of experiments that randomly compared a set of query components against

a small number of library components for different threshold and signifi cance values. The overall match values obtained via the *QoMym* algorithm for the different threshold and signifi cance values were compared against a manual benchmark that we had setup prior to running the experiments. We then gradually added more library components from different domains to determine if the selected threshold and signifi cance values would hold or would need to be adjusted.

There are four major threshold parameters: *label* and *signature* thresholds at the primitive level, as well as *micro* and *macro* thresholds at the micro and macro levels respectively. Figure 8 - 9 depicts the average precision and recall for different label and signature thresholds. Here, the results were obtained by comparing the labels and signatures of a set of query components against those of the components in repository. High precision and recall values are good indicators of the quality of the match. Thus, based on these results, we determined the optimal label threshold to be in the range {`0.7 -- 0.9`}, and the optimal signature threshold to be in {`0.6 -- 0.8`}.

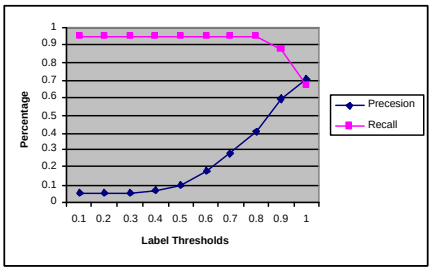

Fig. 8. Variance in the precision and recall obtained for different label threshold values. The label threshold value is represented on the X-Axis, and the percentage of precision and recall is depicted on the Y-Axis.

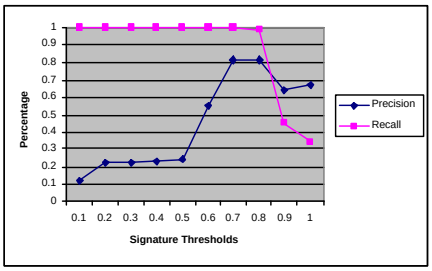

Fig. 9. Variance in the precision and recall obtained for different signature threshold values. The label threshold value is represented on the X-Axis, and the percentage of precision and recall is depicted on the Y-Axis.

Next, to determine the optimal values for the signifi cance parameters of label and signature, we fi xed the label and signature threshold values, and compared the precision and recall obtained by varying the signifi cance values and micro thresholds. Figure 10 shows the precision and recall obtained by the algorithm for different signifi cance values [7] and micro thresholds. We found that for obtaining optimal precision and recall, the signifi cance of the signature should be in the range {`0.1 -- 0.2`}, while the micro thresholds should be in the range {`0.6 -- 0.8`}.

Finally, to determine the optimal macro thresholds, we fi xed the signifi cance value and the micro threshold, and compared the precision and recall obtained by varying the macro thresholds as shown in Figure 11. We found the optimal macro threshold to be in the range {`0.8 -- 0.9`}.

[7] We show signature significance, but the results can be interpreted for label significance value (label = 1 - signature).

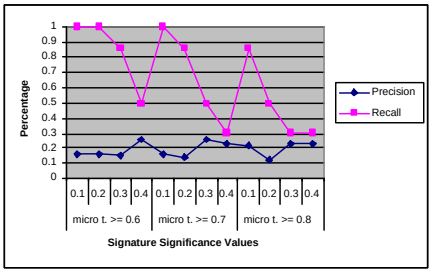

Fig. 10. Variance in the precision and recall obtained for different signature significance values as well as the micro thresholds. Label and signature thresholds were kept constance at `0.8` and `0.7` respectively.

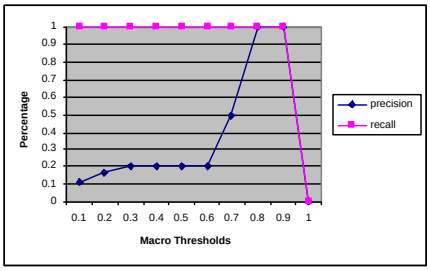

Fig. 11. Variance in the precision and recall obtained for different macro values. Signature significance values and micro thresholds were kept constant at `0.2` and `0.7` respectively.

In the subsequent experiments, we fix the label threshold for the `QoMym` algorithm at `0.8`, the signature threshold at `0.7`, the micro threshold at `0.7`, the macro threshold at `0.8`, and the significance values of label and signature at `0.2` and `0.8` respectively.

6.3 QoMym Match Quality

Based on the threshold and significance values determined in Section 6.2, we ran a set of experiments to evaluate the accuracy of the *QoMym* algorithm. We did this by comparing *QoMym* with our manual benchmark as well as with other state-of-the art algorithms found in literature, namely a signature, linguistic, and a filtering algorithm that uses signature matching as a filter prior to applying a linguistic match algorithm. For these experiments, we selected the `Calendar` component shown in Figure 12 as the query component, and compared it to the components in the repository. Figure 13 shows the results in terms of the number of qualified components returned from the library by the different algorithms for the query component `Calendar`.

```
Component Calendar {
    void setDay (int)                    int getDay ();
    void setMonth (int);                 int getMonth ();
    void setYear (int);                  int getYear ();
    void setStyle (int);                 int getStyle ();
    void setLocale (java.util.Locale);   java.util.Local getLocale ();
}
```

Fig. 12. The Query Component - `Calendar`.

Here the expected number of hits for the `Calendar` component were `4`. The *QoMym* algorithm performed the best returning exactly `4` components, while the signature-based algorithm performed the worst and returned `43` qualified components. It was in-

teresting to note that the filtering algorithm also did better than linguistic and signature algorithms alone returning 6 qualified components. We found there was no difference in the qualified components returned (in this case) if the order of filtering was varied (that is linguistic followed by signature and vice versa).

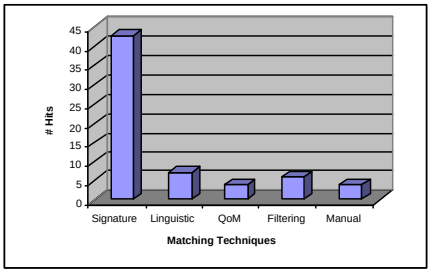

Fig. 13. The Number of Qualified Components Returned For the Different Matching Techniques.

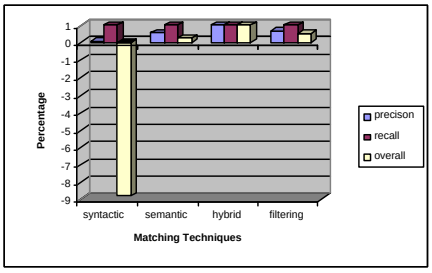

Fig. 14. The Match Quality of the Different Matching Techniques.

Figure 14 depicts the precision and recall of the the different algorithms together with their overall quality. In general, high precision and recall are indicators of high overall quality of the match algorithms. We found that all algorithms had high recall (1.0), that is all algorithms were able to find the expected qualified components. The algorithms however varied in their precision with signature based algorithm having low precision to the *QoMym* algorithm having the best precision. The filtering algorithm shows that the combination of linguistic and signature algorithms can result in higher precision than if the algorithms were used individually. This is in step with the intuitive argument that has been made before [22, 6, 4].

While filtering is a good first step toward combining different algorithms, the preliminary experimental results presented here indicate that a disciplined hybrid combination of the two (and in the future more) algorithms can result in higher overall quality of the retrieved qualified components. It is mainly due to the fact that the significance values are employed to weigh the importance of these algorithms. This allows QoMym to not only discover matches that may have been missed but to also reject matches that are discovered by the filtering technique as different algorithms work independently.

7 Conclusions

The QoMym algorithm offers many advantages over the previously developed component matching approaches. For example, while previous approaches take into account the *method signature* they often discount the importance of labels[8] and the semantic

[8] The method label should in general provide semantic information to partially characterize the methods if component developers implement components by following the software development guide.

information imparted by the same. In our work we now exploit not only the semantic information in labels, but also the syntactic and semantic information contained in properties, event, and the design that are intrinsic parts of a component. In fact we find that with the combined use of the semantic and syntactic information we are able to achieve higher precision and recall without the performance overhead associated with approaches like specification matching. Moreover, our preliminary results suggest that a disciplined combination of different algorithms (linguistic and signature) can provide better overall quality than a naive filter-based approach.

References

1. Dietrich Birngruber. CoML: Yet Another, But Simple Component Composition Language. In *Workshop on Composition Language*, 2001.
2. Don Box. *Essential COM*. Addison-Wesley Publishing Company, 1998.
3. Kajal T. Claypool, Vaishali Hegde, and Naiyana Tansalarak. QMatch: A Hybrid Match Algorithm for XML Schemas. In *Proceedings of the 2nd International Workshop on XML Schema and Data Management (to appear)*, April 2005.
4. Joseph Goguen, Doan Nguyen, Jose Meseguer, Luqi, Du Zhang, and Valdis Berzins. Software Component Search. *Journal of Systems Integration*, 6(1/2):93–134, March 1996.
5. Thomas Gschwind, Johann Oberleitner, and Mehdi Jazayeri. Dynamic Component Extension to Support Cross-Platform Development. Technical Report TUV-1841-2002-19, Technische Universitt Wien, 2002.
6. Jun-Jang Jeng and Betty H. C. Cheng. Specification Matching for Software Reuse: A Foundation*. In *Proceedings of the 1995 Symposium on Software reusability*. ACM Press, 1995.
7. Y. Matsumoto. A Software Factory: An Overall Approach to Software Production. In P. Freeman, editor, *Tutorial: Software Reusability*. IEEE Computer Society Press, 1987.
8. A. Mili, R. Mili, and R. Mittermeir. Storing and retrieving software components: a refinement based system. In *Proceedings of the 16th international conference on Software engineering*, pages 91–100, 1994.
9. G.A. Miller. Wordnet: A Lexical Database for English Language. cogsci.princeton.edu/ wn/, 2002.
10. Hans Muller and Mark Davidson. JavaBeans Specification: Getting Listeners from JavaBeans. http://java.sun.com/products/javabeans, 1996.
11. Frank Pilhofer. Writing and Using CORBA Components. http://www.cs.indiana.edu/ srikrish/orals/mico-ccm.pdf, 2002.
12. R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, 1987.
13. Bill Roth. An Introduction to Enterprise JavaBeans Technology. http://java.sun.com/products/ejb, 1998.
14. J. Siegel. *CORBA: Fundamentals and Programming for the 21st century*. John Wiley, New York, 1996.
15. Vijayan Sugumaran and Veda C. Storey. A semantic-based approach to component retrieval. *ACM SIGMIS Database*, 34(3):8–24, 2003.
16. Naiyana Tansalarak and Kajal T. Claypool. QoM: Qualitative and Quantitative Schema Match Measure. In *Proceedings of the 22nd International Conference on Conceptual Modeling (ER 2003)*, October 2003.
17. Naiyana Tansalarak and Kajal T. Claypool. QoMym: The QoM-based Hybrid Match Algorithm. Technical Report 2004-009, Department of Computer Science, University of Massachusetts - Lowell, August 2004. Available at http://www.cs.uml.edu/techrpts/reports.jsp.

18. Naiyana Tansalarak and Kajal T. Claypool. XCM: A Component Ontology. In *Workshop on Ontologies as Software Engineering Artifacts joint with the 19th Annual ACM Conference on Object-Oriented Programming, Systems, Languages, and Applications*, 2004.
19. Igot Tatarinov and Stratis D. Viglas. Storing and Querying Ordered XML Using a Relational Database System. In *Proceedings of the 2002 ACM SIGMOD international conference on Management of data*, pages 204 – 215. ACM, June 2002.
20. A. Vallecillo, J. Hernandez, and J. Troya. Component Interoperability. Technical Report ITI-2000-37, Departmento de Lenguajes y Ciencias de la Computacion, University of Malaga, July 2000. Available at http://www.lcc.uma.es/∼av/Publicaciones/00/Interoperability.pdf.
21. Amy Moormann Zaremski and Jeannette M. Wing. Signature Matching: a Tool for Using Software Libraries. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. ACM Press, 1995.
22. Amy Moormann Zaremski and Jeannette M. Wing. Specification Matching of Software Components. In *ACM Transactions on Software Engineering and Methodology (TOSEM)*. ACM Press, 1997.