Using Relaxed Class Diagram Intersection to Ease Object-Oriented Library Selection

Technical Report UW-CSE-98-05-02

Amir Michail and David Notkin
Dept. of Computer Science and Engineering
University of Washington, Box 352350
Seattle, WA 98195-2350
{amir,notkin}@cs.washington.edu

Abstract

A key step in software reuse is selecting from among a set of potential reuse candidates. Selecting from among a collection of object-oriented libraries is a costly and difficult activity, in part because they are usually quite large. Current approaches for library selection include informal rules of thumb and complete analysis methods (such as SAAM), all of which require the developer to inspect each library independently.

In this paper, we present a tool-based approach in which a complete collection of candidate libraries are simultaneously compared and contrasted. Specifically, we (1) extract a relaxed class diagram from each library, capturing key structures and also standardizing class and member names, (2) compute a relaxed intersection for each subset of the relaxed class diagrams to capture essential similarities and differences among the libraries, and (3) compute a lattice over the collection of relaxed intersections.

By browsing this lattice, developers can begin assessing the libraries in combination immediately rather than after analyzing each library independently. Our approach can facilitate the assessment process by (1) guiding the developer towards important functional and non-functional properties; (2) helping the developer compare and contrast these properties across libraries; and (3) providing access to the relevant source code in each library for further inspection. All aspects of our approach are supported by CodeWeb, a tool we have built for assessing C++ and Java libraries.

1 Introduction

Selecting from among a set of candidates is an important step in software reuse [7]. There are a number of approaches for selecting from among relatively small reuse candidates such as individual methods or classes [3, 4, 9, 12]. These approaches are generally query-based, which usually works well in part because there are often a reasonably large number of candidates.

But selecting among larger candidates — such as selecting a thread library or a user-interface framework — is difficult and potentially very expensive [10].[1] Indeed, Sparks, Benner and Faris give this advice for framework selection:

> "Budget adequately to support frameworks. Expect the evaluation and selection of a framework to take up to six staff-months per new framework." [10, p. 54]

The selection among larger candidates introduces a different set of problems than those for small reuse candidates. First, non-functional properties — extensibility, flexibility, understandability, testability, etc. — are increasingly important. Second, there are usually a relatively small set of library candidates (perhaps a dozen or so), at least in part because the cost of producing libraries is necessarily higher than that of producing smaller candidates. Third, libraries are often pre-selected (for instance, the developer may manually gather a handful of user in-

[1] To simplify the presentation, we use the term *library* to mean any large candidate, including libraries, frameworks, applications, etc.

terface frameworks to assess), precisely because they ostensibly provide the basic functional behavior that the developer wants; therefore, comparing and contrasting structures of the libraries is likely to be more important than with smaller candidates, where specific functional behavior is usually more important. Fourth, the libraries are simply larger: each may have 100 or more classes, making it impractical to compare libraries at the source level (even if there are only a few under consideration). Moreover, superficial differences between them can easily confuse tools; for example, even a minor difference in naming (say, between DocumentView and DocView) might cause a tool to find a difference rather than a similarity between libraries, and, as another example, minor differences in the inheritance structures might lead to a similar confusion.

To address these issues, we have developed an approach for abstracting the essence of the structure of the libraries, providing a tool that allows the developer to browse among the structural similarities and differences among them. We present four specific steps and results:

- Given a library, we describe how to compute its *relaxed class diagram*. The intent of a relaxed class diagram is to capture the key structures among the classes in a library as well as to standardize class and member names. This heuristic computation is designed to emphasize essential structures in the library while abstracting away from minor and potentially distracting details.

- Given a collection of relaxed class diagrams from a set of libraries, we describe how to compute the *relaxed intersection*, or matching, of these diagrams. The intent of the relaxed intersection is to capture where libraries (represented by their relaxed class diagrams) are essentially similar to and different from other candidates.

- Given relaxed intersections for every subset of the relaxed class diagrams for a set of libraries, we create a lattice that represents where specific combinations of libraries are similar and different, and how.

- Our tool, CodeWeb, allows the programmer to view a specific relaxed intersection and to browse the lattice and the associated source code in a variety of ways.

Section 2 discusses related work. Section 3 presents the computation of relaxed class diagrams. Section 4 presents the definition of relaxed intersection across a set of relaxed class diagrams. Section 5 describes how to compute a lattice over the relaxed intersections. Section 6 discusses the tool and shows, by example, how it can be used to infer functional and non-functional properties. Section 7 summarizes the work, concluding with a number of open questions.

2 Related Work

The current state of the art in selecting among library candidates relies on qualitative assessment. This may take the form of informal tips for selecting frameworks [10] or a complete analysis method, such as SAAM [5]. Either way, the developer manually inspects each library, reads the documentation, examines the architecture, considers subjective scenarios, and other available information.

Tool support for candidate selection has been lacking. Although there are tools that help a developer examine an individual library in terms of architecture, style, etc. [1, 6, 8, 11], we know of no tools that help the developer directly compare several libraries. With existing tools, the user must manually integrate the knowledge learned about each library.

In contrast, there are numerous reuse tools that help a developer select a fine-grained component (such as a function or class) in a particular library. These include tools that use free-text indexing [4], facets [9], signature matching [12], and formal specifications [3].

As noted above, such techniques do not carry over to library selection because: (a) the libraries under consideration usually have similar functional but different non-functional properties, while the components in a particular library typically have different functional but similar non-functional properties; and (b) there are usually only a few libraries under consideration (typically less than ten) but there may be hundreds or thousands of components within a particular library. Consequently, tools for selecting components in a particular library are typically query-based (since browsing hundreds or thousands of components is not practical) and usually take into account functional properties (since that's what the user is most interested in).

Our approach does not and is not intended to replace qualitative methods for assessing libraries. Rather, it facilitates such methods by providing the developer easy access to similarities and differences among the library architectures as well as the relevant source fragments in each library (which allows the developer to infer functional and non-functional properties about the libraries). The developer need no longer consider each library in isolation (as with manual inspection or with existing tools).

3 Relaxed Class Diagrams

The first step in our approach is to analyze each library, producing a *relaxed class diagram*, which is intended to capture the essence of the class diagram for that library in a way that makes finding similarities across libraries easier. (Although relaxed class diagrams are each a function of one library, the diagrams are not designed to facilitate comparison *within* a single library — but rather, to make it easier to find similarities *across* libraries.)

Producing a relaxed class diagram consists of two basic activities: standardizing names and extracting structure.

3.1 Standardizing Names

Not all programmers use the same naming conventions. For example, one programmer might write document_view while another would use DocumentView. Libraries often add a prefix to the name, as in wxb_DocumentView. To complicate matters further, programmers may use abbreviations as with wxb_DocView. This variation makes it difficult to compare the class diagrams of different libraries.

We standardize class and member names in three ways by: (1) handling upper and lower case in a consistent way; (2) expanding common abbreviations; and (3) getting rid of unnecessary prefixes. In particular, we define two heuristic functions, Υ_i and Φ_i, which yield the standard name of a class and a member in library i, respectively.

Consider a class or member name. First, we perform the following two steps on each name:

- split the name x into words $w_1 - \cdots - w_n$ where word boundaries are inferred from underscores and/or capital letters (e.g., split class wxb_DocView into wxb-Doc-View and member wx_GetPos into wx-Get-Pos); and
- change each w_j to w'_j where w'_j standardizes case and expands any common abbreviations (e.g., we change wxb-Doc-View to Wxb-Document-View and wx-Get-Pos to Wx-Get-Position).

Next, we determine the *maximal prefix of words,* $w'_1 - \cdots - w'_m$ of $w'_1 - \cdots - w'_n$, where $0 \leq m < n$, such that

- $w'_1 - \cdots - w'_m$ occurs as a prefix of this library's class (or member) names at least $f(N)$ times, where N is the total number of classes (or members) in the library and f is some heuristic function; and
- each word w'_i in the prefix has length $\leq k$ for some constant k.

Intuitively speaking, it is not likely that a prefix contributes meaningfully to a name if it occurs in many other names. In practice, we find that using $f(N) = \max(3, \min(10, 0.1N))$ and $k = 3$ yields good results. Returning to our example, suppose that this procedure results in a Wxb prefix in Wxb-Document-View and a Wx-Get prefix in Wx-Get-Position.

At this point, the procedure for standardizing class and member names diverges so we describe each separately in what follows.

3.1.1 Class Names

We would like to rename $w'_1 - \cdots - w'_n$ to $w'_{m+1} - \cdots - w'_n$ by stripping the (possibly empty) prefix $w'_1 - \cdots - w'_m$. However, we would also like Υ_i to be a bijection to avoid merging the members of two separate classes in library i that have similar names. So we will not necessarily strip all prefixes.

Now, a maximal prefix that occurs more frequently than another maximal prefix is more likely to be a standard prefix in the library. So, we strip a prefix $w'_1 - \cdots - w'_m$ if and only if the prefix $w'_1 - \cdots - w'_m$ occurs more frequently than any other maximal prefix $w''_1 - \cdots - w''_q$ of a class $w''_1 - \cdots - w''_q - w'_{m+1} - \cdots - w'_n$. (By definition, an empty prefix occurs infinitely many times.)

For example, we rename Wxb-Document-View to Document-View if and only if no other maximal prefix $w''_1 - \cdots - w''_q$ of a class $w''_1 - \cdots - w''_q$–Document-View occurs as frequently as Wxb. If renaming occurs, Υ_i(wxb_DocView) yields DocumentView. Otherwise, Υ_i(wxb_DocView) yields the original name wxb_DocView.

If we assume that the class names in library i are distinct, then one can show that Υ_i is a bijection.

3.1.2 Member Names

Unlike with class names, we do not care if two original member names map to the same new member name.

Instead, we are more interested in having member names match across libraries. Indeed, we do not distinguish member functions and member variables (so a member function and member variable can have their names mapped to the same name), and we perform the following steps:

- always strip the prefix (e.g., strip the prefix Wx-Get from Wx-Get-Position to get Position); and
- strip any remaining standard prefixes such as Get, Set, etc. (e.g., if the prefix were only Wx then this step would strip out the following Get).

Now consider the remaining words in the member name (e.g., Position in our example). To avoid clutter in the relaxed class diagram, we ignore the member completely if more than one word remains. In our experience, multi-word members tend not to be as "fundamental" to the class as single word members. In any case, multi-word members are not likely to match across different libraries.

Moreover, even if only one word remains, we ignore the member if that word is in the *stop list*. The stop list contains common members, such as those which copy or destroy objects, which tend to say very little about what a particular class does (as they are common to so many classes).

Returning to our example, Φ_i(wx_GetPos) would yield Position. Unlike Υ_i which is a bijection, Φ_i may be partial and not one-to-one.

3.2 Extracting Structure

Given a library i and renaming functions Υ_i and Φ_i, we show how to compute the corresponding relaxed class diagram C_i. The diagram C_i is a directed graph in which the nodes denote classes and members while the edges denote direct/indirect inheritance, reference, and membership relationships between them.

Classes: $\{a\} \in C_i$ iff there is some class a' in library i such that $\Upsilon_i(a') = a$.

(The reason for the singleton set notation for classes will become apparent in Section 4.) Observe that we abstract out any properties of a' so that it is more likely that a class matches across libraries. For example, we don't care whether a' is an abstract class (i.e., type/interface) or concrete class.

Of course, it is possible to keep this distinction and allow matching across libraries by introducing three kinds of classes: "abstract only", "concrete only", and "abstract/concrete". When comparing several libraries, we would say that an "abstract/concrete" class a is shared among them if a is abstract in some libraries and concrete in some others.

However, we do not see this distinction as important enough to warrant a more complicated notation for representing classes that match across libraries since that would make it harder for the user to interpret the results. (We shall also abstract out member properties for the same reason.)

Direct Inheritance: $\{a\} \Rightarrow \{b\} \in C_i$ iff there are classes a' and b' in library i such that $\Upsilon_i(a') = a$, $\Upsilon_i(b') = b$, and a' directly inherits from b' in library i.

Direct Reference: $\{a\} \mapsto \{b\} \in C_i$ iff there are classes a' and b' in library i such that $\Upsilon_i(a') = a$, $\Upsilon_i(b') = b$, and a' directly references b' (i.e., a' has a member variable of type b' or pointer to b') in library i.

Observe that we don't distinguish composition from reference. This is done because: (1) it makes matching across libraries more likely (without complicating the notation, as mentioned earlier); and (2) it is generally not possible to determine whether a reference is a composition.

Members: $m_{\{a\}} \in C_i$ iff there is a class a' and member m' in C_i such that $\Upsilon_i(a') = a$, $\Phi_i(m') = m$, and a' defines or declares member m' in library i.

Direct Membership: $\{a\} \rightarrow m_{\{a\}} \in C_i$ iff $m_{\{a\}} \in C_i$.

A member m cannot appear in isolation but must be associated with some class $\{a\}$ using the $m_{\{a\}}$ notation. The subscript $\{a\}$ on m distinguishes different definitions/declarations of m in C_i.

Suppose a' and b' both define m', a' directly inherits from b', $\Upsilon_i(a') = a$, $\Upsilon_i(b') = b$, and $\Phi_i(m') = m$. In such a case, $\{a\} \rightarrow m_{\{a\}}$, $\{b\} \rightarrow m_{\{b\}}$, and $\{a\} \Rightarrow \{b\}$ all appear in C_i. Just as m' is overridden in library i, it also overridden in C_i (as indicated by the two instances of m and the inheritance relationship between $\{a\}$ and $\{b\}$ in C_i).

As explained above, we do not distinguish member functions from member variables. Moreover, we abstract out any properties of m'. For example, we don't care in what ways (if any) m' is accessible to clients and base classes (e.g., private, protected, or public) and we also don't care whether or not m' is abstract, virtual, constant, and/or static. Again, this allows more member matches across libraries without complicating the notation.

Unique Membership: $\{a\} \leftrightarrow m_{\{a\}} \in C_i$ iff $\{a\} \rightarrow m_{\{a\}} \in C_i$ and *only* classes $\{b\}$ that inherit (directly or indirectly) from $\{a\}$ define $m_{\{b\}}$ (in which case $m_{\{b\}}$ overrides $m_{\{a\}}$).

Observe that $\{a\} \leftrightarrow m_{\{a\}} \in C_i$ implies $\{a\} \rightarrow m_{\{a\}} \in C_i$. Unique membership edges are used as an indicator that m's behavior is somehow specific to a class $\{a\}$ in the library (modulo any methods that override m in descendents of $\{a\}$). We say that m is the *unique member of* $\{a\}$. Unique members can be used to match classes with different names as discussed in Section 4.

We can increase the likelihood that some relationships match across libraries by considering not only *direct* relationships but also by considering the *closure* of these relationships. However, we keep a distinction between the two since we believe the user would want to know when relationships are direct or not (even at the cost of a more complicated notation).

We add closure edges for inheritance, reference, and membership paths in the relaxed class diagram C_i as follows.

Closure Inheritance: $\{a\} \Rightarrow^+ \{b\}$ appears in C_i iff there is a path of length at least one from $\{a\}$ to $\{b\}$ over inheritance edges ($\Rightarrow$) in C_i.

For example, if $\{a\} \Rightarrow \{b\}$ and $\{b\} \Rightarrow \{c\}$ are in C_i, then $\{a\} \Rightarrow^+ \{b\}$, $\{b\} \Rightarrow^+ \{c\}$, and $\{a\} \Rightarrow^+ \{c\}$ are also in C_i.

Closure Reference: $\{a\} \mapsto^+ \{b\}$ appears in C_i iff there is a path from a to b over inheritance ($\Rightarrow$) and reference edges ($\mapsto$), that includes at least one reference edge, in C_i.

For example, if $\{a\} \Rightarrow \{b\}$, $\{b\} \mapsto \{c\}$, and $\{c\} \Rightarrow \{d\}$ are in C_i, then $\{a\} \mapsto^+ \{c\}$, $\{a\} \mapsto^+ \{d\}$, and $\{b\} \mapsto^+ \{c\}$, $\{b\} \mapsto^+ \{d\}$ are also in C_i.

Closure Membership: $\{a\} \rightarrow^+ m_{\{b\}} \in C_i$ iff $\{b\} \rightarrow f_{\{b\}} \in C_i$ and either $\{a\} \Rightarrow^+ \{b\} \in C_i$ or $a = b$.

For example if $\{a\} \Rightarrow \{b\}$, $\{b\} \Rightarrow \{c\}$, and $\{c\} \rightarrow m_{\{c\}}$ are in C_i, then $\{a\} \rightarrow^+ m_{\{c\}}$, $\{b\} \rightarrow^+ m_{\{c\}}$, and $\{c\} \rightarrow^+ m_{\{c\}}$ are also in C_i.

4 Relaxed Intersection

In Section 3, we have shown how to determine the relaxed class diagram C_i of a library i. In this section, we define the *relaxed intersection* of a set of relaxed class diagrams $\{C_{j_1}, \ldots, C_{j_k}\}$, which we denote by $I_{\{j_1,\ldots,j_k\}}$. We use the same node and edge notation in the relaxed intersection graph as that used in the individual relaxed class diagrams. Moreover, we define the intersection so that $I_{\{i\}} = C_i$.

4.1 Class Matching

We assume that classes with the same name in different libraries are likely to serve similar purposes, so they should appear in the intersection.

Of course, it may be the case that the libraries use completely different names to denote classes with similar functionality. For example, class `task` in one library may be similar to `thread` in another. In such a case, we would want to match `task` with `thread` and use the *class set* {`task`, `thread`} to denote the match in the relaxed intersection.

We match classes structurally based on their contents. In particular, if several classes, each in a different library, share a unique member (see Section 3.2), then this indicates that the member's behavior is specific to those classes and no others. This is a strong hint that these classes may have a similar purpose in each library. For example, we might match `task` with `thread` because in each library we have a class, either `task` or `thread`, with unique member `yield()`.

In what follows, we denote class sets by uppercase letters and individual classes by lowercase letters. Class matches and unique membership edges are determined in the intersection as follows.

Unique Membership: $A \leftrightarrow m_A \in I_X$ iff

- for all $a \in A$, $\{a\}$ occurs in at least one C_i where $i \in X$;
- for all $i \in X$, exactly one of the classes a in A occurs in C_i; and
- for all $a \in A$ and $i \in X$, if $\{a\}$ appears in C_i then $\{a\} \leftrightarrow m_{\{a\}}$ appears in C_i.

Classes: $A \in I_X$ iff

- $A = \{a\}$ and $\{a\} \in C_i$ for all $i \in X$; or
- $A \leftrightarrow m_A \in I_X$ for some m.

C_1	C_2	$I_{\{1,2\}}$
$\begin{tikzcd} \{e\} \arrow[r] & m_{\{e\}} & \\ & \{x\} \arrow[r] & m_{\{x\}} \\ \{b\} \arrow[r, Rightarrow] & \{d\} \arrow[u, Rightarrow] & \\ \{a\} \arrow[u, Rightarrow] \arrow[r, Rightarrow] & \{c\} \arrow[u, Rightarrow] \arrow[r] & m_{\{c\}} \end{tikzcd}$	$\begin{tikzcd} \{y\} \arrow[r] & m_{\{y\}} & & \\ \{d\} \arrow[u, Rightarrow] & & \{e\} \arrow[r] & m_{\{e\}} \\ \{b\} \arrow[u, Rightarrow] \arrow[r] & m_{\{b\}} & & \\ \{c\} \arrow[u, Rightarrow] & & & \\ \{a\} \arrow[u, Rightarrow] \arrow[r] & m_{\{a\}} & & \end{tikzcd}$	$\begin{tikzcd} \{e\} \arrow[r] & m_{\{e\}} & \\ \{b\} \arrow[r, Rightarrow] & \{d\} \arrow[r, "+"] & m_{\{d\}} \\ \{a\} \arrow[u, Rightarrow, "+"] \arrow[r, Rightarrow] & \{c\} \arrow[u, Rightarrow, "+"] \arrow[r, "+"] & m_{\{c\}} \end{tikzcd}$

Figure 1: Preserving overriding relationships among members. (We do not show the redundant closure edges.)

Observe that if classes a and b both occur in some library i, then they will not be matched in an intersection of libraries that includes i. The reason is that since library i has both classes a and b, there is evidence against the hypothesis that these classes serve the same function.

Also, observe that we show all potential matches in the intersection. For example, it may be that a matches b based on unique member m and yet a also matches c based on unique member m'. In such a case, we leave it up to the user to determine which (if any) of the matches is correct.

In the definitions that follow, it will be convenient to map a class set $A = \{a_1, \ldots, a_k\}$ in the relaxed intersection I_X to the singleton class set $\{a_j\}$ (where $a_j \in A$) that appears in a particular relaxed class diagram C_i:

- $\Psi_X(A, i) = \{a_j\}$ iff
  - $A = \{a_j\}$ and $\{a_j\} \in I_X$; or
  - $A \leftrightarrow m_A \in I_X$ for some m and $\{a_j\} \leftrightarrow m_{\{a_j\}} \in C_i$.

4.2 Inheritance, Reference, and Membership Relationships

Inheritance and reference edges, whether direct or closure, occur in the relaxed intersection if and only if they occur in each of the relaxed class diagrams in that intersection. More precisely, we have the following for direct inheritance and reference.

Direct Inheritance: $A \Rightarrow B \in I_X$ iff $\Psi_X(A,i) \Rightarrow \Psi_X(B,i) \in C_i$ for all $i \in X$.

Direct Reference: $A \mapsto B \in I_X$ iff $\Psi_X(A,i) \mapsto \Psi_X(B,i) \in C_i$ for all $i \in X$.

And for closure inheritance and reference we have:

Closure Inheritance: $A \Rightarrow^+ B \in I_X$ iff $\Psi_X(A,i) \Rightarrow^+ \Psi_X(B,i) \in C_i$ for all $i \in X$.

Closure Reference: $A \mapsto^+ B \in I_X$ iff $\Psi_X(A,i) \mapsto^+ \Psi_X(B,i) \in C_i$ for all $i \in X$.

Our rule for determining whether a member m appears in the relaxed intersection or not is complicated by several factors: (1) we want to associate each member with a class set (e.g., m_A) to distinguish different declarations/definitions of m; (2) we want to take into account inheritance of members; and (3) we want to preserve member overriding relationships across libraries.

Formally, by "overriding" we mean the following:

- m_A *overrides* m_B iff $A \Rightarrow^+ B$.

To understand how we preserve overriding relationships in the intersection, consider all class sets $\{A_1, \ldots, A_r\}$ in I_X that define/inherit a *particular* member m in C_i for all $i \in X$. If one includes all the direct/closure inheritance relationships among these class sets in I_X (as defined above), then we have one or more directed acyclic graphs (DAGs) $G_1, \ldots, G_s$ in I_X induced by the A_j's and inheritance relationships among them. (See $I_{\{1,2\}}$ in Figure 1 which shows two such DAGs.)

In each G_k, a "root" A_j doesn't inherit from any other class set that defines/inherits m in I_X. (See "roots" $\{d\}$ and $\{e\}$ in Figure 1.) Since $\Psi_X(A_j, i)$ defines/inherits m in each C_i but A_j can't inherit m

from any class set in I_X, we associate m with A_j by placing $A_j \rightarrow^+ m_{A_j}$ in I_X.

In each G_k, a "non-root" A_j inherits from some other class set in I_X that defines/inherits m. (See "non-roots" $\{a\}, \{b\}$, and $\{c\}$ in Figure 1.) Should $A_j \rightarrow^+ m_{A_j}$ appear in I_X? If so, then this implies that A_j defines/inherits an m that overrides that defined/inherited by its ancestors in G_k. We preserve the member overriding relationship in the following sense: $A_j \rightarrow^+ m_{A_j}$ appears in I_X if and only if for every C_i, $\Psi_X(A_j, i)$ defines/inherits *some* m that overrides *all* m's defined/inherited by $\Psi_X(A_j, i)$'s ancestors in C_i that are also its ancestors in G_k. More formally,

Members: $m_A \in I_X$ iff for all $i \in X$ there exists an s_i such that

- $\Psi_X(A, i) \rightarrow^+ m_{\{s_i\}} \in C_i$; and
- for any ancestor B of A in I_X and any t_i such that $\Psi_X(B, i) \rightarrow^+ m_{\{t_i\}} \in C_i$, we have $\{s_i\} \Rightarrow^+ \{t_i\} \in C_i$ (i.e., $m_{\{s_i\}}$ overrides $m_{\{t_i\}}$ in C_i).

Closure Membership: $A \rightarrow^+ m_B \in I_X$ iff $m_B \in I_X$ and either $A \Rightarrow^+ B \in I_X$ or $A = B$.

Finally, we define the direct membership edges.

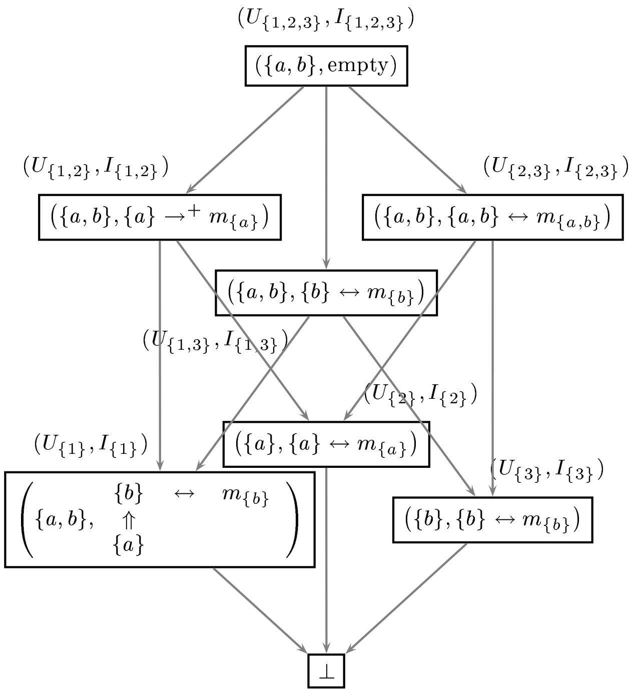
Direct Membership: $A \rightarrow m_A \in I_X$ iff $\Psi_X(A, i) \rightarrow m_{\Psi_X(A,i)} \in C_i$ for all $i \in X$.

5 Lattice of Relaxed Intersections

In Section 4, we showed how to compute a relaxed intersection I_X of a set of libraries X. However, as the number of libraries grows, the libraries share fewer class diagram fragments. This means that I_X may contain few (if any) shared elements among a large set of libraries X.

Moreover, computing a single relaxed intersection I_X only tells us what all the libraries X share in common and does not tell us anything about similarities and differences among subsets of X.

Consequently, we not only compute the relaxed intersection of X but also the relaxed intersection of every subset of X. We then place the relaxed intersections in a lattice which users can browse to identify similarities and differences among any subset of libraries in X [2].



$(U_{\{1,2,3\}}, I_{\{1,2,3\}})$
$(\{a,b\}, \text{empty})$
$(U_{\{1,2\}}, I_{\{1,2\}})$
$(\{a,b\}, \{a\} \rightarrow^+ m_{\{a\}})$
$(U_{\{2,3\}}, I_{\{2,3\}})$
$(\{a,b\}, \{a,b\} \leftrightarrow m_{\{a,b\}})$
$(\{a,b\}, \{b\} \leftrightarrow m_{\{b\}})$
$(U_{\{1,3\}}, I_{\{1,3\}})$
$(U_{\{2\}}, I_{\{2\}})$
$(U_{\{1\}}, I_{\{1\}})$
$(\{a\}, \{a\} \leftrightarrow m_{\{a\}})$
$(U_{\{3\}}, I_{\{3\}})$
$\left(\{a,b\}, \begin{array}{ccc} \{b\} & \leftrightarrow & m_{\{b\}} \\ \Uparrow & & \\ \{a\} & & \end{array}\right)$
$(\{b\}, \{b\} \leftrightarrow m_{\{b\}})$
$\perp$

Figure 2: A sample lattice. (We do not show the redundant closure edges in the relaxed intersections.)

In particular, each lattice node consists of a pair (U_Y, I_Y) where set U_Y keeps track of all the classes defined by one or more libraries in Y (and so can be used as an index as we explain in Section 6):

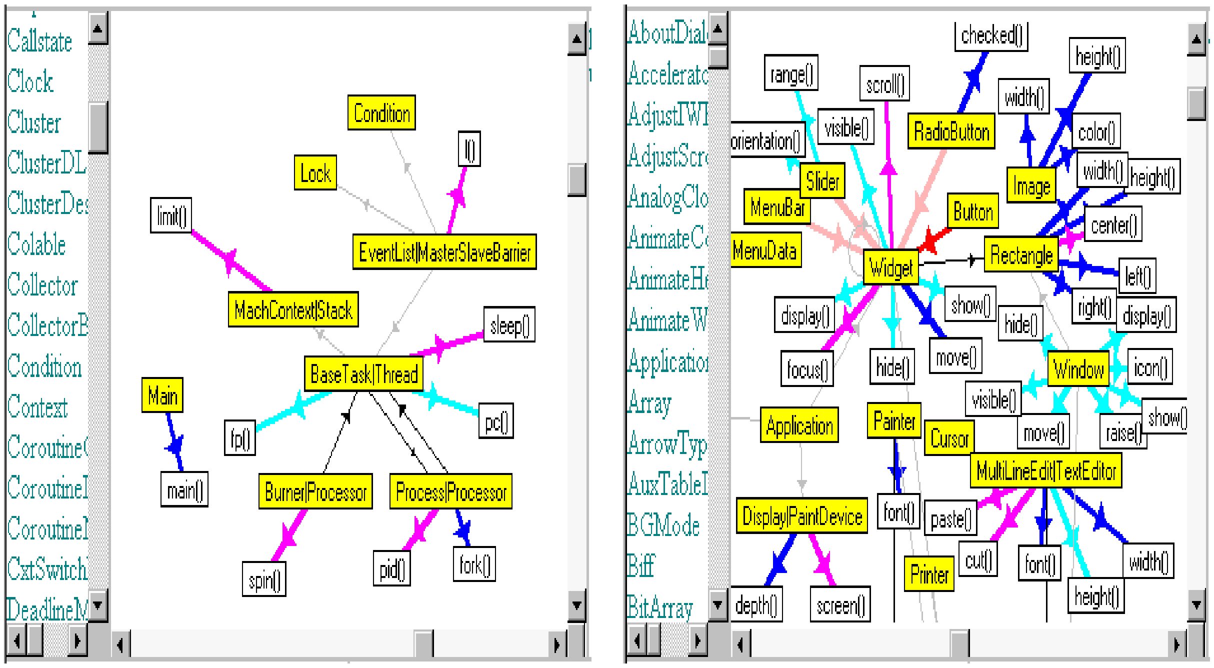
- $a \in U_Y$ iff there is an $\{a\} \in C_i$ for some $i \in Y$.

There is a node (U_Y, I_Y) in the lattice for every subset Y of X (including the empty set). Although this means the lattice has $2^{|X|}$ nodes, this is not a serious problem since one often compares fewer than ten libraries in a particular domain. Moreover, as we mention in Section 6, our tool has several user interface features that allow one to quickly move to interesting nodes in a large lattice.

The nodes are arranged in the lattice in the following way:

- node (U_Y, I_Y) is an ancestor of (U_Z, I_Z) if and only if $Y \supset Z$.

Intuitively speaking, (U_Y, I_Y) appears as an ancestor of (U_Z, I_Z) in the lattice if and only if (U_Y, I_Y) is more "general" than (U_Z, I_Z). (See Figure 2.) Moreover, the more general (U_Y, I_Y) represents a superset of the libraries in (U_Z, I_Z).



(a) Thread Libraries: Presto-1.0 and μC++-4.6

(b) GUI Libraries: Qt-1.2 and JX-1.0.3

Figure 3: The top node in the thread and GUI lattices.

6 Tool

All aspects of our approach are supported by CodeWeb, a tool we have built for assessing C++ and Java libraries. Given a set of libraries, the tool automatically performs the steps described in Sections 3, 4, and 5 to build the lattice over the collection of relaxed intersections.

We have run CodeWeb on simulation, thread, and GUI libraries, with as many as eight libraries at a time. To illustrate the kinds of similarities and differences that CodeWeb identifies among libraries, and how one might use them to infer functional and non-functional properties, we demonstrate the tool on two C++ thread libraries, Presto 1.0 and μC++ 4.6, and two C++ GUI libraries, Qt 1.2 and JX 1.0.3.

CodeWeb shows exactly one node of the lattice on the screen at a time. In particular, Figure 3 shows the top node (U_Y, I_Y) of the thread and GUI lattices in parts (a) and (b), respectively. (The top node represents the relaxed intersection of two libraries in each case; both lattices contain four nodes, one for each subset of the libraries.) Observe that the set of classes in U_Y is shown on the left while the relaxed intersection I_Y is shown graphically on the right.

CodeWeb represents classes in shaded rectangles while their members appear, with a "()" suffix, in unshaded rectangles. Matched classes are denoted with the "|" symbol. Reference and inheritance relationships between classes are shown by narrow and wide edges, respectively. Direct and indirect relationships are indicated by dark and light shading, respectively.

Clicking on a class (or matched classes) in I_Y displays the corresponding source definition for each library $i \in Y$. For example, clicking on BaseTask|Thread shows the source for BaseTask in μC++ and Thread in Presto. Clicking on a class in U_Y moves from the current node to the most general lattice node whose relaxed intersection contains that class. Thus, one can view U_Y as an index for descendent nodes. For example, clicking on BaseTask moves to the child node representing μC++.

CodeWeb also provides two other facilities to simplify browsing (which are not shown in Figure 3): (1) a list of lattice nodes with a rank associated with each one to indicate how "interesting" it is—where interesting means that it contains a substantial number of class diagram fragments shared by many libraries; and (2) "graphical deltas" from the current node to each parent and to each child which make it easier to see how parents and children differ from the current node.

6.1 Inferring Functional Properties

One can often find important functional concepts in a domain by browsing the nodes near the top of the lattice. In the thread example, the top node has BaseTask|Thread which indicates a match between classes BaseTask and Thread as inferred from the unique member sleep(). Indeed, the names "task" and "thread" are often used interchangeably in thread libraries and are central concepts to that domain. The top node also shows classes for two important synchronization primitives: condition variables and locks.

Similarly, important functional concepts are also shown in the GUI example. For example, the top node has Window, Widget, Button, MenuBar, Image, and Painter. Observe that Button and MenuBar inherit from Widget in both libraries. One can also see key members in a class. For example, Window has show(), hide(), visible(), move(), raise(), display(), and icon(). Finally, observe that MultiLineEdit|TextEditor indicates a correct match between the classes MultiLineEdit and TextEditor as inferred from the unique members cut() and paste().

The lattice also provides a way to compare and contrast functional properties across libraries. For example, further exploration of the lattice nodes would reveal that μC++ has extensive real-time facilities while Presto does not. In particular, one could easily see real-time classes such as RealTimeTask, RealTimeCluster, PeriodicBaseTask, SporadicBaseTask, and DeadLineMonotonic which are not in Presto.

CodeWeb provides another way to compare and contrast functional properties: by clicking a class in a relaxed intersection, one can see its source code definition in each library in the intersection. For example, by clicking on the Painter class in the GUI example, one can easily compare the drawing primitives (which are members of Painter) that are supported by each library. One might determine, for instance, that while both Qt and JX support lines, rectangles, ellipses, and arcs, only Qt supports Bezier curves.

6.2 Inferring Non-Functional Properties

CodeWeb can also help a developer infer non-functional properties from a library such as extensibility, adaptability, modularity, flexibility, understandability, maintainability, testability, etc. These properties are particularly important with respect to the fundamental domain classes (and the relationships between them). Consequently, one can use CodeWeb to (1) identify these classes and relationships (as discussed in Section 6.1) and (2) inspect the corresponding source code fragments in each library to infer non-functional properties (as we shall discuss below).

Consider extensibility. Clearly, it is desirable to be able to easily extend and/or modify the functionality of classes in various ways. For example, in the GUI libraries, one can compare how easy it is to create new widgets by inspecting the source code of classes Button, RadioButton, and Slider, all of which inherit from Widget in both JX and Qt.

One way to gauge extendibility is to look for virtual member functions in the libraries. Virtual member functions indicate "hot-spots", or areas that can be altered in descendent classes (but at the cost of performance). CodeWeb's ability to compare source code side-by-side for classes in different libraries can be very useful in this regard.

As an example, comparing the source for the Thread class in Presto and the BaseTask class in μC++ shows that almost every key member function in Thread (such as start(), run(), wakeup() and sleep()) is declared virtual while this is not the case for similar member functions in BaseTask. Further examination of the libraries shows that while Presto has virtual member functions sprinkled liberally throughout, the authors of μC++ have used more restraint and only put virtual member functions where they thought was absolutely necessary.

On the other hand, μC++ uses more finely-grained classes than Presto, and these classes are arranged in a more elaborate and deeper inheritance hierarchy. For example, whereas Thread inherits from Object in Presto, BaseTask inherits from BaseCoroutine which in turn inherits from MachContext. Consequently, one can more easily extend classes in μC++ without inheriting unneeded baggage (assuming of course that one doesn't run into the virtual member problem discussed earlier).

As another example, consider the GUI libraries JX and Qt. If one clicks on the Button class to inspect the corresponding source in each library, one would notice that both libraries use implicit invocation to provide loose coupling between objects. It is well known that this improves adaptability, understandability, and maintainability.

Further investigation of the source code shows that Qt allows one to connect a member that "emits" a signal to other members that will receive it. JX uses a more coarse-grained approach: one connects objects rather than object members. However, Qt uses a preprocessor to implement its mechanism, which may cause problems with support tools that require valid C++ source.

7 Conclusions and Future Work

In this paper, we have described a tool-based approach for library selection. Our approach is not intended to replace qualitative methods for assessing libraries but, rather, to complement them by providing easy access to similarities and differences among the library class diagrams as well as the relevant source fragments in each library.

We have demonstrated — using our tool CodeWeb — how a developer might infer functional and nonfunctional properties from a set of libraries. For future research, we plan to conduct extensive user testing to see how useful our approach is in practice.

We have observed that CodeWeb may also be useful for evolution and slicing. For example, one may run the tool on different versions of a library to browse its evolutionary history. It is also possible to run the tool on two libraries A and B, where A is more general than B, to extract a "slice" of the narrower domain of B from A (and thus learn about that aspect of A). It would be interesting to explore evolution, slicing, and other potential uses of the tool further.

From our research, it appears that class diagrams are an appropriate high-level abstraction for comparing libraries. However, it may also be interesting to consider collaboration diagrams, sequence diagrams, state diagrams, etc. (although these would be harder to extract automatically from the source).

Also, one may define other forms of relaxed class diagrams and relaxed intersection that are useful for purposes other than selection. For example, we are currently working on a variation that allows a developer to compare how several sample applications reuse a particular library. This would then help the developer reuse that library in a new application.

Acknowledgments

We would like to thank Will Tracz and Michael Ernst for valuable feedback on this research.

References

[1] T. J. Biggerstaff. Design Recovery for Maintenance and Reuse. *Computer*, 22(7):36–49, 1989.

[2] C. Carpineto and G. Romano. A Lattice Conceptual Clustering System and Its Application to Browsing Retrieval. *Machine Learning*, 24(2):95–122, 1996.

[3] P. Chen, R. Hennicker, and M. Jarke. On the Retrieval of Reusable Software Components. In *2nd International Workshop on Software Reusability)*, pages 99–108. IEEE, 1993.

[4] W. B. Frakes and B. A. Nejmeh. Software Reuse through Information Retrieval. In *20th Hawaii International Conference on System Sciences*, pages 530–535. IEEE, 1987.

[5] R. Kazman, L. Bass, G. Abowd, and M. Webb. SAAM: A Method for Analyzing the Properties of Software Architectures. In *16th International Conference on Software Engineering*, pages 81–90. IEEE, 1994.

[6] R. Kazman and S. J. Carriere. View Extraction and View Fusion in Architectural Understanding. In *5th International Conference on Software Reuse*. IEEE, 1998.

[7] C. W. Krueger. Software Reuse. *ACM Computing Surveys*, 24(2):131–183, 1992.

[8] G. C. Murphy, D. Notkin, and K. Sullivan. Software Reflexion Models: Bridging the Gap between Source and High-Level Models. In *3rd ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 18–28, 1995.

[9] R. Prieto-Diaz and P. Freeman. Classifying Software for Reusability. *IEEE Software*, 4(1):6–16, 1987.

[10] S. Sparks, K. Benner, and C. Faris. Managing Object-Oriented Framework Reuse. *Computer*, 29(9):52–61, 1996.

[11] A. S. Yeh, D. R. Harris, and M. P. Chase. Manipulating Recovered Software Architecture Views. In *Proceedings of the International Conference on Software Engineering*, pages 184–194, 1997.

[12] A. M. Zaremski and J. M. Wing. Signature Matching: A Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146–170, 1995.