An Objective Reuse Metric: Model and Methodology

Yih-Farn Chen and Balachander Krishnamurthy and Kiem-Phong Vo

AT&T Bell Laboratories, 600 Mountain Avenue, Murray Hill, NJ 07974 USA.

Abstract. Software reuse is an effective way to gain productivity in constructing software systems. In order to continuously monitor the progress of reuse in the context of a project, we need an *objective* and *repeatable* way to measure the extent of reuse. This paper proposes a model and methodology to automatically compute a general objective reuse measure from the source code of applications and reusable software repositories. Unlike consumer-oriented reuse measures in the literature, this measure is useful from the viewpoints of both reuse consumers and reuse producers. The former can use it to estimate the extent of reuse in their applications while the latter can use it to determine the impact of certain reusable components. In contrast to other objective code-based measures, our measure is both *complete* and *precise* in that it is calculated based on and only on those code entities that are essential to the functioning of an application. While the model and methodology are largely language-independent, we have implemented tools to compute the reuse measure from C code. We report experiences from using the measure on a few large software systems and discuss its use in discovering and encouraging reuse.

1 Introduction

Software reuse is a good way to gain productivity but measuring its extent and impact is still a controversial subject. This is because reuse involves many dimensions not all of which are straightforward to estimate. For example, consider the effect of standardizing the look and feel in a family of applications. This type of interface reuse has direct and substantial impact on the requirements and design phases of building a new application. However, since such requirements are often only informally stated and then only in relatively imprecise natural language documents, it is not obvious what unit of measurement would apply. Many other dimensions affecting reuse are based on factors such as time, process, management, and people. These are not easy to measure objectively.

One aspect of reuse that can be measured objectively is code reuse. Here, the relevant question to a consumer of reusable code is: *How much of my application code did not have to be written because of reuse?* A precise answer to this question would give a reliable indicator as to how reuse helped to reduce development effort. Conversely, the relevant question for a producer of reusable code is: *How much of my reusable component is actually reused in some given set of applications?* A precise answer to this question would help in finding out where reuse had the most impact and in making decisions to allocate resources in managing the creation, enhancement and maintenance of reusable code. To answer these questions, one needs to define what is meant by code, code constituting an application, code that is reused, and finally select some reasonable unit to measure code. Then, reuse measures and measuring methods can be defined based on the choices made. This paper proposes a model and methodology to address these questions.

The rest of this paper is organized as follows. Section 2 defines a language-independent model of a software system in terms of its basic code entities and a reuse measure that is easily computable from the code entities. Section 3 specializes the measure to C code and gives examples of how it is applied to a few large software systems. Section 4 compares our measure to others proposed in the literature. Section 5 discusses some future work including finding ways to employ the measure in discovering, encouraging, and managing software reuse. Section 6 concludes the paper.

2 Model and measure

2.1 Software system model

The first step in working with a software system is to fix a notion of its composition in terms of the implemented code. Following the approach outlined in [CRV94], we assume that a software system S is composed of some set E of basic code entities. Code entities may depend on one another in the sense that using an entity e may imply that other entities must be present so that e will be meaningful. Let D be the set of pairwise direct dependencies among entities, i.e., (e, f) is in D if e is directly dependent on f. We also assume that some subset R of E forms a set of roots so that each execution or use of S must start from some element in R. In this way, the software system S is modeled as a directed graph (R, E, D) with root set R, node set E and edge set D. We shall use the notations S_R, S_E and S_D to denote respectively the sets R, E, and D of the software system S. Two properties are required of S:

- *Completeness*: Each executable or usable part of the software system is covered by some entity in E.
- *Reachability*: Each entity in E is reachable from some root entity in R.

Let's examine the above conditions in the setting of a program written in the C language. One possible choice of entities would be: types, macros, functions, and variables. Here, the direct dependency between two entities simply means that one's definition requires the other's (for example, a variable requires a type in its declaration). The root set of a C application consists of the single function `main()` and the entity set all entities reachable from `main()`. Since executable code must be contained in functions and memory areas must be defined by variables, it is clear that the set of defined entities is complete. On the other hand, the source code of the program may contain extraneous code such as code left over from debugging. Such code, often referred to as dead code, is explicitly excluded by the reachability condition.

Code entity types must be carefully chosen to suit the processing purposes. For example, for automatic system construction using a build tool such as *nmake* [Fow90], it may be convenient to pick source files as basic entities. In the C language, one type of dependency among source files would be file inclusion. That is, a file must have all the header files that it includes to be meaningful. Another type of dependency is based on code reference. That is, if some code fragment (for example, a function) defined in a file A requires a code fragment in another file B (for example, calling a function in B), then A depends on B. Given these two types of file dependency, it is not hard to see that the completeness and reachability conditions are satisfiable for file entities. However, for the purpose of measuring software reuse, file entities are too coarse as a model of entity type because there is much information in a file (for example, comments or dead code) that is irrelevant to the functioning of the system but cannot be excluded at that level.

Once a set of code entity types have been chosen, we can associate with each entity e a weight, $weight(e)$. Then, the size of S is defined to be:

$$Size(S) = \sum_{e \in E} weight(e)$$

For example, if the weight function is unity for all entities, then the above expression simply counts the entities. If lines of code are used as weights, then the above expression computes the total number of lines of code that constitute the given system.

2.2 Measuring code reuse

To measure reuse, the code constituting an application system must be classified into two types: *application code* or code implementing specific application features, and *reused code* or code belonging to some reusable code repository. Note that a reusable repository p is a software system on its own and its external interface is defined by the root set p_R. Let S be an application system reusing p. We define:

- $Refer(S,p)$: the set of entities in p that S directly refers to. So $Refer(S,p)$ is simply the intersection of S_E and p_R. For example, if functions are the only entity type, then $Refer(S,p)$ would be the set of functions in p that S calls directly.
- $Closure(S,p)$: the set of entities in p that S directly or indirectly refers to via the entity dependencies. Thus $Closure(S,p)$ is the intersection of S_E and p_E. Again, if functions are the only entity type, then $Closure(S,p)$ includes all functions in p that are directly or indirectly called by S.

In practice, because of dead code possibility, the entity sets S_E and p_E may be available only as subsets of some larger entity sets. However, they can be easily computed from the respective dependency graphs and root sets using any standard graph reachability algorithm such as depth-first or breadth-first search[AHU87].

When multiple reusable repositories are used, there may be interdependencies among entities in all repositories. Let P be a collection of repositories and p one of its members. By overloading notation, we define:

- $Refer(S, P)$: the union of all $Refer(S, p)$, i.e., the set of entities in all the libraries of P that the application system S directly refers to.
- $Closure(S, p, P)$: the set of entities in the library p that are in $Refer(S, p)$ or indirectly used by some member of $Refer(S, P)$.
- $Closure(S, P)$: the union of all $Closure(S, p, P)$.

To make these concepts clear, consider Figure 1 that shows an example of two application systems S_1, S_2 reusing two repositories p_1, p_2. We shall also use the application and repository names to refer to the corresponding collections of entities. The combinations of Closure values are

- $Closure(S_1, p_1, P) = e_4, e_5$
- $Closure(S_1, p_2, P) = e_6$
- $Closure(S_2, p_1, P) = e_5$
- $Closure(S_2, p_2, P) = e_7, e_9$

Note that while S_1 does not use e_6 in p_2 directly, e_6 is reachable indirectly through e_4 in p_1 and is therefore considered reused by S_1.

Now we can define the *Reuse* measure of a system S with respect to a repository or a collection of repositories:

$$Reuse(S, p) = \sum_{e \in Closure(S,p)} weight(e)$$

$$Reuse(S, P) = \sum_{e \in Closure(S,P)} weight(e)$$

Now if P consists of all the reusable repositories that S uses, we can define the extent of code reuse in the system S as:

$$Reuse(S) = Reuse(S, P)$$

In practice, it is useful to look at percentage of code that is reused. For this purpose, from an application point of view, we can define the *application reuse percentage*:

$$AppReusePerc(S) = \frac{Reuse(S)}{Size(S)}$$

From a reusable code supplier point of view, it is useful to know the percentage of a repository that is reused in some application, so we define the *repository*

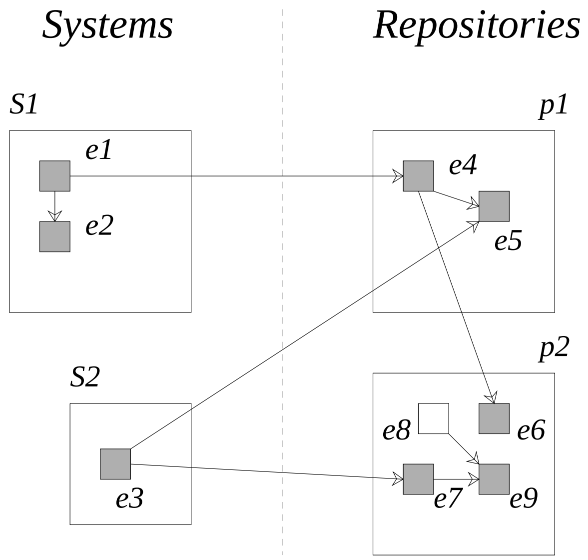

Fig. 1. Two applications reusing repositories p_1 and p_2

reuse percentage for a repository p with respect to some application system S as:

$$RepReusePerc(S,p) = \frac{Reuse(S,p)}{Size(p)}$$

Using the same example in Figure 1 with the trivial weight function of unity value for each entity, we obtain the following:

– $Reuse(S_1) = 3$, $Reuse(S_2) = 3$
– $AppReusePerc(S_1)$=60%, $AppReusePerc(S_2)$=75%
– $RepReusePerc(S_1, p_1)$=100%, $RepReusePerc(S_1, p_2)$=25%
  $RepReusePerc(S_2, p_1)$=50%, $RepReusePerc(S_2, p_2)$=50%

In this way of measuring reuse, each application is considered a distinct *consumer* with a well-defined *organization boundary* as discussed by Poulin, Caruso, and Hancock in [PCH93]. The reuse measures can vary significantly if we change the organization boundaries. For example, if we consider S_1 and S_2 to be a single application S, then the reuse numbers will change to the following:

– $Reuse(S)$=5
– $AppReusePerc(S)$=62.5%
– $RepReusePerc(S, p_1)$=100%, $RepReusePerc(S, p_2)$=75%

A few observations about the reuse measure $Reuse(S)$ can be made:

- $Reuse(S)$ does not include any multiplicity of usage even if an entity is reused multiple times from the application and/or other repositories. If an entity is reused, it is counted exactly once.
- $Reuse(S)$ does not include any entity in the application system S that is not in some reusable repository.
- $Reuse(S)$ only includes those entities in repositories actually used by the application.

Thus, $Reuse(S)$ provides a precise method to measure external reusable code actually used by S. In the next section, we discuss its use in the C language.

3 A reuse measure for C programs

To apply the size and reuse measures described in Section 2 for the C programming language, we choose the C code entities: types, functions, macros, global and static variables. These entities, their attributes and their interdependencies are readily available from a program database produced by CIA, the C Information Abstraction System[CNR90][Che95]. As discussed in Section 2, these entities are complete in the sense that they account for every possible executable code fragment. Here, it is convenient to use libraries as instantiations of reusable repositories. The root entities of a library are simply the types, functions, macros and global variables that are advertised as part of the external interface.

Next we present studies in estimating the reuse extents in two large sets of application programs: one from our department, the AST tools and libraries, and one from a large communications software project. In these studies, each program is considered a separate application system with a single root entity, the `main()` function. At least for the AST collection, this is a reasonable choice as the programs are developed independently by different researchers to solve rather different types of problems. We shall use two different entity weight functions. The W_e function assigns a weight of 1 for each entity, i.e., it does entity counting. The W_s function weighs each entity by its number of non-commented source lines, i.e., it counts NCSL.

3.1 The AST tools and libraries

Our department produces and maintains a large collection of tools and libraries commonly referred to as AST[Kri95], Advanced Software Technology. The code was written by about a dozen researchers over a period of many years and enjoys wide use both within our department and across AT&T. In fact, some components are available world-wide on many platforms.

Reuse can be studied from two perspectives: how an application uses reusable libraries and how a library is used in the reusing applications. Figure 2 shows the dependency relationships among several AST commands and the libraries they use. For example, *app* and *ciacom* depend on *libpp* for preprocessing C code and all tools directly or indirectly depend on *libast* for various functions.

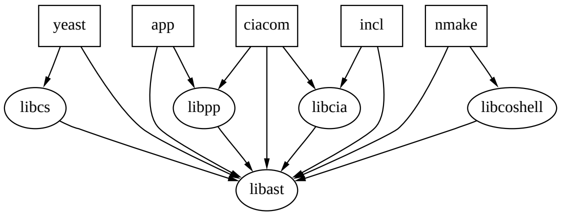

Fig. 2. The library dependency graph of several AST components

Table 1 shows the sizes of a few AST libraries measured in the number of entities and NCSL. The libraries shown are: *libcia* to perform IO on the CIA database of entities and relationships, *libpp* to perform C preprocessing in the style of the C preprocessor, *libcoshell* to run shell commands on different networked machines, *libcs* to establish connections and manage message passing among heterogeneous networked servers, and *libast*, the largest AST library composed for many smaller packages to provide common functionality such as POSIX and ANSI-C compliance routines to enhance portability, safe and fast buffered IO [KV91], memory allocation, etc.

Library	Entities	NCSL
libcia	70	473
libpp	1242	12430
libcoshell	190	1770
libcs	859	9816
libast	4838	36272

Table 1. Size metrics of a subset of AST libraries

Table 2 shows the sizes and reuse extents of a few tools in the AST collection: *app*, an annotation preprocessor to insert programmable assertions into C code [Ros92], *ciacom*, the compiler for the C Information Abstractor, *incl*, a program to analyze header file hierarchies and determine unnecessary header files [VC92], *nmake*, a program to configure and build software systems [Fow90], and *yeast*, an event-action matching system [RK95]. Despite the wide range of functionality, the reuse percentages for these tools are quite high ranging from from 43.7% for *nmake* to a high of 82.3% for *incl*. This means that reuse has been effective in reducing the work needed to develop our tools.

Table 3 shows the library reuse percentages between various pairs of tool and library as measured in the number of entities and NCSL. The library reuse percentages for *libast* are relatively small (typically less than 20%) because except for basic functionality such as IO and memory allocation, different commands use

Tool	Entities	NCSL	*Reuse* with W_e	*AppReusePerc* with W_e	*Reuse* with W_s	*AppReusePerc* with W_s
app	715	5881	1251	63.6%	7970	57.5%
ciacom	840	5327	1482	78.2%	10458	66.2%
incl	131	1085	547	80.7%	5055	82.3%
nmake	961	17319	1408	59.4%	13427	43.7%
yeast	842	7288	1552	64.8%	7625	51.1%

Table 2. Size metrics of a subset of AST tools

different parts of the library. An exception is *nmake* where the *libast* library reuse percentage is 34%. This is because many components in *libast* are built during the implementation of *nmake*. On the other hand, the library reuse percentages for *libcia* are high relative to both *incl* and *ciacom* because these commands rely exclusively on *libcia* for IO on the CIA database.

Reusing command	Reused library	*Reuse* with W_e	*RepReusePerc* with W_e	*Reuse* with W_s	*RepReusePerc* with W_s
app	*libast*	596	12.3%	4405	12.1%
ciacom	*libast*	773	16.0%	6567	18.1%
incl	*libast*	402	8.3%	4723	13.0%
nmake	*libast*	1318	27.2%	12324	34.0%
yeast	*libast*	1347	27.8%	5078	14.0%
app	*libpp*	655	52.7%	3565	28.7%
ciacom	*libpp*	651	52.4%	3556	28.6%
ciacom	*libcia*	58	82.9%	335	70.8%
incl	*libcia*	55	78.6%	332	70.2%
nmake	*libcoshell*	90	47.4%	1103	62.3%
yeast	*libcs*	205	23.9%	2547	35.3%

Table 3. Reuse metrics of several AST library and command pairs

3.2 A communications software project

To try our measure on code not familiar to us, we studied a subsystem of a large communications software project. This subsystem is organized into two reusable libraries, *libcore* and *libdomain*, and a set of application programs. The library *libcore* deals with core functionalities of the subsystem while *libdomain* deals with communications to external components. Tables 4 shows the sizes of the two libraries.

Table 5 shows the application reuse percentages of the application programs. The application reuse percentages of *create* and *gen* are quite low. On further examination of the source code, we see that *gen* is a table generator that manipulates a number of private translation tables to generate others that are of more

Library	Entities	NCSL
libdomain	45	374
libcore	244	2247

Table 4. Size metrics of the two libraries in a communications software project

interest project-wide. The large amount of code for such manipulations explains the low reuse percentages.

Command	Entities	NCSL	*Reuse* with W_e	*AppReusePerc* with W_e	*Reuse* with W_s	*AppReusePerc* with W_s
audit	23	557	51	68.9%	1518	73.2%
delete	7	61	24	77.4%	384	86.3%
export	50	1285	89	64.0%	1667	56.5%
create	32	657	14	30.4%	105	13.8%
gen	220	2363	20	8.3%	179	7.0%
query	211	1709	119	36.1%	1989	53.8%

Table 5. Size and reuse metrics of a set of applications in a communications software project

Table 6 shows the library reuse percentages between pairs of applications and libraries. A missing command and library combination indicates that the command does not reuse the library. The division of functionalities between the two libraries is reflected in the commands. For example, *audit*, *create*, and *export* require many external interactions so they depend heavily on *libdomain*. On the other hand, *delete*, *gen*, and *query* provide functionality internal to the subsystem so they do not use *libdomain* in any way.

Reusing command	Reused library	*Reuse* with W_e	*RepReusePerc* with W_e	*Reuse* with W_s	*RepReusePerc* with W_s
audit	*libcore*	42	17.2%	1301	57.9%
create	*libcore*	8	3.3%	8	0.4%
delete	*libcore*	24	9.8%	384	17.1%
gen	*libcore*	20	8.2%	179	8.0%
query	*libcore*	119	48.8%	1989	88.5%
export	*libcore*	61	25.0%	1426	63.5%
audit	*libdomain*	9	20.0%	217	58.0%
create	*libdomain*	6	13.3%	97	25.9%
export	*libdomain*	28	62.2%	241	64.4%

Table 6. Library reuse metrics of a subsystem of a communications software project

3.3 Discussion

Though our experiences in the above two studies are still preliminary, certain aspects of reuse measures are clear. We discuss these aspects below.

- *Objectivity*: This means the ability for different people to estimate the amount of reuse consistently. Objectivity of a measure is good because it increases the trustworthiness of the measure by minimizing variations due to personal biases. Being code-based, our measure is objective. This has proved to be an important factor in estimating the reuse extent of the AST tools which were developed at different times by different people with different levels of experiences in using the libraries. It would have been hard to control variations in the measure if aspects such as ease of use or economical benefit were included. It is certainly easier to agree on the choice of NCSL or entity count as an entity weight function than, say, estimating the number of hours saved by using some reusable code entities.
- *Precision*: This means the ability to pinpoint the exact amount of reuse. Precision of a measure is good because it increases the reliability of the measure by reducing variations due to packaging or implementation. Our measure is precise in two different ways. First, it defines a set of code entities at a small enough granularity to track closer to the code execution semantics. Second, it uses reachability to determine only those entities that are actually reused. This has allowed us to provide believable estimates of reuse in our AST tools. It is worth noting that we started our studies with a measure similar to that proposed in [PCH93] and [AE92]. This measure estimated the reuse level of the annotation preprocessor *app* at close to 90% because it reuses *libast*, a large library. This is counter-intuitive to both the library and tool researchers. Our NCSL measure estimates the reuse level of *app* at 57.5%, a more believable figure.
- *Applicability*: One measure of goodness for any tool is the range of its applicability. By virtue of being objective and precise, our measure is potentially a good tool in the management of a reuse program. Though still largely unexplored, aspects of the measure can help with decisions concerning the management of assets. For example, the library reuse percentage *RepReusePerc* can indicate library packages with larger reuse and thus deserving more maintenance resources. In a case like *libpp* where the measure based on entity count differs significantly from the measure based on NCSL, it is an indication that some entities have unusually large sizes. This may be useful information in any effort to reengineering the code. More on the controversial side but sometimes useful is the use of reuse measures to draw certain conclusions about the practices in a reuse program. For example, in the AST work, we actively promote the development of reusable components as a part of research into new and interesting applications [FKV95]. It would be nice to have a concrete means to evaluate the goodness of this approach. At the risk of sounding self-serving, the relatively high reuse percentages in our tools and libraries shown in Section 3.1 do indicate that we are on the right track.

4 Related work

In this section we look at related work in the field with specific answers to the questions raised in Section 1 for the producers and consumers of reusable components. While comprehensive economic reuse models such as those proposed by Gaffney and Durek [GD89] and Lim [Lim94] help us appreciate the nature of the trade-offs among the costs and benefits of reuse, they require relative cost figures of integrating and developing a reusable component, which may be difficult to obtain. We are mainly interested in reuse measures that can be computed automatically and objectively from the source code repository of any software organization. Several papers share, at least informally, our goal and focus on code metrics; however, there are critical differences between our approach and theirs in several areas.

The first area concerns the *granularity* of entities used in computing reuse measures. Poulin et al [PCH93] consider all shipped source instructions of a reusable component as being reused. This is complete in the sense discussed in Section 2 but like the file entity type, it is too coarse and cannot exclude unreachable code. Agresti and Evanco [AE92] consider *reuse fraction* as a simple proportional measure based on the fraction of reused compilation units. Similar to the measure defined by Poulin et al, this measure is complete but too coarse. Devanbu and Karstu [DK94] define their measures based on functions in C++ code. These measures are not complete since they leave out essential code entities such as types, macros and variables. We define a general framework to develop measures that are both complete and precise. Our proposed measure for the C language includes all essential type, variable, function and macro entities in reusable libraries but it excludes any entities not reachable from the application system.

The second area concerns the inclusion of internal application code as reused code. For example, Frakes and Terry [FT94] consider any function in the application code that is used more than a certain preset threshold as being internally reused. Though there may be some tenuous arguments that such functions have the potential of becoming reusable components, Favaro[Fav91] points out that there is a need for the software engineering community to appreciate the intrinsic technical difficulty of making reusable components that can be easily integrated into real applications. We believe that reuse should be measured based on code known to be reusable and has been reused in multiple applications. Besides, internal reuse can lead to counter-intuitive results. For example, consider recursive functions. Should a recursive call be considered an instance of reuse or is that just the way that the function is implemented? What if a recursive function is reimplemented using an iterative method for efficiency? In fact, doing this may decrease the reuse value since function calls are being eliminated. To avoid such logical inconsistencies and to have a measure that estimates only actual reuse, our measure excludes consideration for internal reuse.

The third area concerns the use of *multiplicity* of references to a reusable component. Frakes and Terry[FT94] propose that the number of reuses should pass a *threshold level* before a component can be considered reused. In their paper on an axiomatic approach to evaluate reuse measures, Devanbu and Karstu [DK94] also discuss a measure called *expanded code size*, which includes the frequencies of reuse of the same component. Like internal reuse, this type of reuse consideration can lead to counter-intuitive results. Consider the common practice of unrolling a loop for efficiency. The number of calls to a function may increase because of loop unrolling, causing a net gain for the reuse measure although the software system is essentially unchanged. Poulin et al argue that while an *organization* may *use* a component numerous times, it can *reuse* a component but once. This is reasonable so we use the same approach. As we pointed out in Section 2, the root sets of repositories and applications defined in our model correspond to the *organization boundaries* defined in [PCH93]'s model. The reuse numbers can vary based on how the organization boundaries are defined. But this is to be expected.

Our division of applications and reusable repositories follows the Barnes and Bollinger[BB91] producer/consumer model. As the paper points out, one likely major hurdle to software reuse is the lack of incentive strategies to encourage coordinated reuse. While most reuse measures concentrate only on the reuse percentages of the applications on the consumer side, such as the ones proposed in [PCH93], the numbers do not provide sufficient feedback to those producers who design repositories. Our *RepReusePerc* measure can help managers appreciate the extent of reuse of a repository and help allocating resources for further development of existing repositories. We have applied the measure to *disco* (Section 5), a prototype graphical software tool to aid the discovery of reuse, with browsing capabilities and presentation of reuse percentages.

5 Future work

Precise reuse measures are useful in convincing users that components are worthwhile for reusing and in helping repository maintainers to better allocate maintenance resource. Toward this end, we have implemented a prototype tool called *disco* to display reusable functions and reusing entities in a graphical form. The entity and relationship information is obtained from a CIA database.

Figure 3 shows two functions `InsertPrimEv` (belonging to *Yeast*) and `csname` (belonging to *libcs*). A *disco* user comes to such a screen by first looking at the components used by the tool *Yeast*, selecting the edge between *Yeast* and *cs*, then selecting the pair (`InsertPrimEv`, `csname`) from all pairs of reusing and reused functions. Library reuse percentages *RepReusePerc* are used to highlight edges, functions, and libraries using either colors or line thickness. In this way, *disco* allows a repository browser to get a quick idea of the reuse extent of any library or component. Conversely, a repository maintainer can, at a glance, see which parts of the library are heavily used and which are not. Such information can be used as a parameter in allocating maintenance resource or to help library designers in reshaping the library design and its assembly. In future work, we plan to have audio and video clips associated with the nodes of the graph to obtain documentation and other information associated with the application

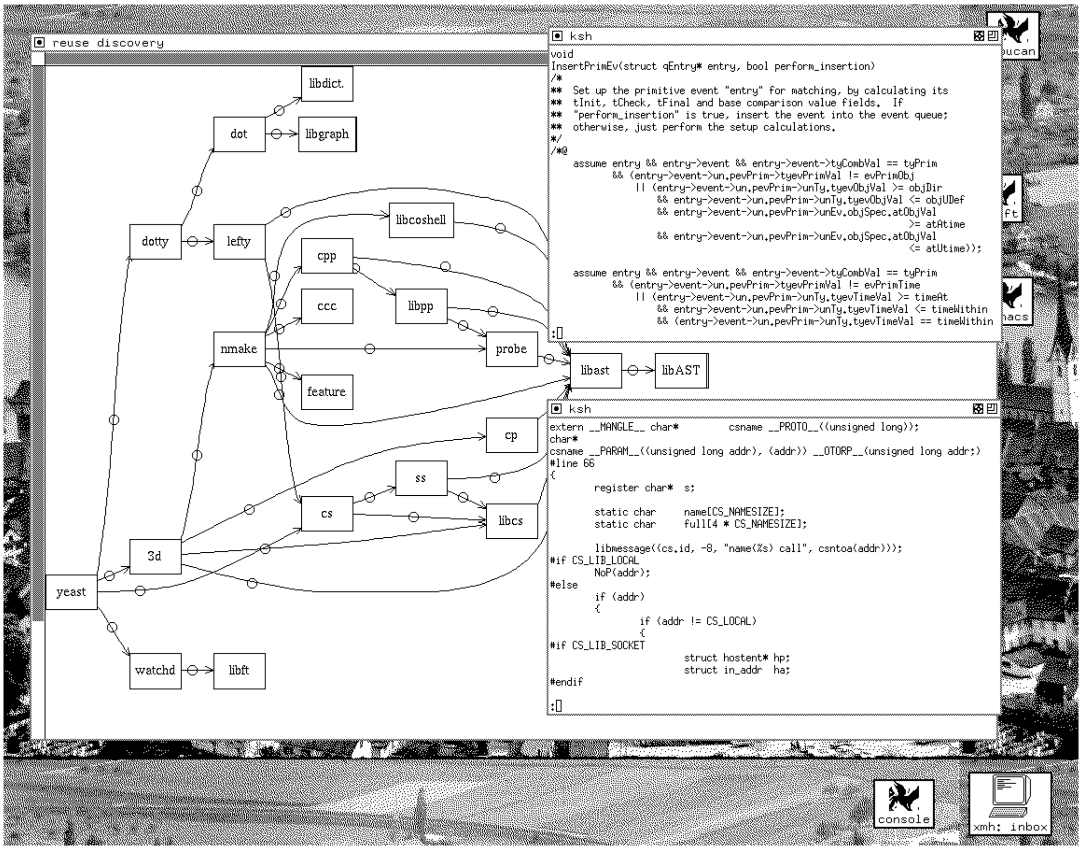

Fig. 3. Function references between *Yeast* and *cs*

and/or the code.

Aside from experimenting with using reuse measures to stimulate further reuse along the lines of tools like *disco*, it would be interesting to learn more about the predictive behavior of a reuse measure. For example, though we do believe that reuse increases productivity, it would be nice to validate this in a real software system implemented in different ways with and without reuse. Unfortunately, it is a rare event that large pieces of software are rewritten just to take advantage of reusable components. However, we do have some examples of this locally in some parts of software systems such as CIA or *yeast*. We shall try some small experiments along this way. Finally, given the multitude of reuse measures proposed in the literature, it may be useful to perform a comparative study of our measures and others using either an axiomatic approach such as the one proposed in [DK94] or experiments on actual code.

6 Summary

We have proposed a model and measure to compute objective reuse values based purely on source code. The measure is designed to be useful to both consumers and producers of reusable code. Consumers of reusable code can use the measure to estimate the extent of reuse in their applications while producers of such code can use the measure to find out the impact of certain components. Many code-based measures proposed in the literature lack either completeness or precision. In contrast, our measure is both complete and precise because not only does it encompass all code entities that may be involved in the execution of an application, it also includes only the ones essential to the functioning of the application. The measure has been implemented for the C language. We report experiences in using the reuse measure on two large software projects. Even at a preliminary stage, the measure does provide us with valuable insights into the software systems examined. The precision of the measure in estimating the true fraction of code from a reusable source is a first step towards providing a tool that can help with discovery and administrative tasks concerning reusable assets. We mentioned the *disco* prototype, a preliminary step along that path. Finally, it should be stressed that all code-based metrics including ours do not and cannot measure reuse benefits from non-functional parts of a reuse program such as portability, ease of use, reduced development costs etc. For example, even though the event processing tool *Yeast* reuses only 35% of *libcs*, the connection service library, the existence of *libcs* was crucial because it let *Yeast* do away with many administrative burdens in writing communications software and made it portable to non-BSD environments. Nonetheless, by virtue of its being objective, precise and automatically computable, our reuse measure should prove useful in the continuous development and assessment of a reuse program.

7 Acknowledgements

We would like to thank Premkumar Devanbu for valuable discussion on reuse measures and his assistance in identifying several related papers.

References

[AE92] W. Agresti and W. Evanco. Projecting software defects in analyzing Ada designs. *IEEE Transactions on Software Engineering*, 18(11):988–997, 1992.

[AHU87] A. Aho, J. Hopcroft, and J. Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1987.

[BB91] B.H. Barnes and T.B. Bollinger. Making Reuse Cost-Effective. *IEEE Software*, 8(1):13–24, 1991.

[Che95] Y.-F. Chen. Reverse engineering. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 6. John Wiley & Sons, New York, 1995.

[CNR90] Y.-F. Chen, M. Nishimoto, and C.V. Ramamoorthy. The C Information Abstraction System. *IEEE Transactions on Software Engineering*, 16(3):325–334, March 1990.

[CRV94] Y.-F. Chen, D.S. Rosenblum, and K.-P. Vo. TestTube: A System for Selective Regression Testing. In *Proceedings of the 16th International Conference on Software Engineering*, 1994.

[DK94] P. Devanbu and S. Karstu. Measuring the Benefits of Software Reuse. Technical report, AT&T Bell Laboratories, 1994. Available from prem@research.att.com.

[Fav91] J. Favaro. What price reusability? A case study. *Ada Letters, Spring 1991*, 11(3):115–124, 1991.

[FKV95] G.S. Fowler, D.G. Korn, and K.-P. Vo. Principles for Writing Reusable Library. In *Proceedings of the ACM SIGSOFT Symposium on Software Reusability (SSR'95)*, Seattle, April 1995.

[Fow90] G.S. Fowler. A Case for make. *Software – Practice and Experience*, 20:35–46, June 1990.

[FT94] W. Frakes and C. Terry. Reuse level metrics. In *Proceedings of 3rd International Conference on Software Reuse*, pages 139–148. IEEE Computer Society, November 1994.

[GD89] J.E. Gaffney and T.A. Durek. Software reuse – key to enhanced productivity: Some quantitative models. *Information and Software Technology*, 31(5):258–267, 1989.

[Kri95] B. Krishnamurthy, editor. *Practical Reusable UNIX Software*. John Wiley & Sons, 1995.

[KV91] D.G. Korn and K.-P. Vo. SFIO: Safe/Fast String/File IO. In *Proceedings of the Summer '91 Usenix Conference*, pages 235–256. USENIX, 1991.

[Lim94] W. Lim. Effects of Reuse on Quality, Productivity, and Economics. *IEEE Software*, 11(5):23–30, September 1994.

[PCH93] J. Poulin, J. Caruso, and D. Hancock. The Business Case for Software Reuse. *IBM Systems Journal*, 32(4):567–594, 1993.

[RK95] D.S. Rosenblum and B. Krishnamurthy. Generalized event-action handling. In B. Krishnamurthy, editor, *Practical Reusable UNIX Software*, chapter 9. John Wiley & Sons, New York, 1995.

[Ros92] D.S. Rosenblum. Towards a method of programming with assertions. In *Proceedings of the 14th International Conference on Software Engineering*, pages 92–104. Association for Computing Machinery, May 1992.

[VC92] K.-P. Vo and Y.-F. Chen. Incl: A tool to analyze include files. In *Summer 1992 USENIX Conference*, pages 199–208, June 1992.

This article was processed using the LaTeX macro package with LLNCS style