Automating The Detection of Reusable Parts in Existing Software

Michael F. Dunn
John C. Knight

Computer Science Report No. TR-92-34
September 17, 1992

Automating The Detection Of Reusable Parts In Existing Software†

Michael F. Dunn *John C. Knight*

Department of Computer Science
University of Virginia
Thornton Hall
Charlottesville, VA 22903

Abstract

We present a model based on an expert-system approach for the scavenging of reusable components from existing software systems. We also describe a toolset called Code Miner that implements part of the model. The toolset uses Prolog as its inference engine. Code Miner is designed to assist the programmer in finding reusable components in existing software written in C.

To investigate the feasibility of the approach we conducted an empirical study of the effectiveness of the toolset. In the study, public-domain software was scanned by the toolset for reusable parts and its output was examined by a team of human experts. We present the results of this experiment.

Keywords: Software reuse, reusable components, reengineering, Prolog, expert system.

† This work sponsored in part by the Virginia Center for Innovative Technology, grant number INF-92-001.

1. Introduction

While a great deal of research over the past several years has been devoted to the creation of reusable software components and component libraries, the issue of how to salvage reusable components from existing systems has remained relatively unexplored. One often hears the comment that reusable components have to be designed rather than discovered, but to a software development manager with access to a large inventory of existing software, such a statement is both intellectually and economically dissatisfying.

Intuitively, it is reasonable to assume that of the several billion lines of source code that have been developed over the years, some significant subset ought to be reusable with little or no modification. Location of reusable parts within existing software systems would be of considerable benefit, however, only if the parts were truly useful and the cost of finding them was less than the cost of rebuilding them. This is an important trade-off, but it is a trade-off that is difficult to make since the value of parts found and the cost of finding them cannot be known in advance.

Given this situation, we decided to investigate the possibility of substantially automated location of reusable parts in the hope that an automated system would yield reusable parts from existing systems but do so with very low cost, mainly just computer time. If this were possible, the yield would not have to be great in order for the technique to have value. In this paper we describe a prototype scavenging toolset, called Code Miner, that is being developed at the University of Virginia. The goal of the toolset is to provide a very high level of assistance to the programmer in identifying potentially reusable parts from an assortment of C programs. The emphasis of the toolset is detecting components with a high probability of being reusable rather than finding all of the possibly reusable components.

Informal code scavenging with little or no tool support has been a common practice among programmers since the early days of the discipline. It is only fairly recently that the activity has undergone serious study. Selby [Sel88] describes reuse of code from existing systems at NASA and identifies a number of characteristics of reused components, such as small size and small numbers of input/output parameters. He identifies 32 percent of the code used in new systems as being reused from existing systems. This significant percentage gives impetus to the argument that code scavenging is a potentially fruitful activity. The scavenging work described by Dunn and Knight [DuK91] also indicates the usefulness of this activity.

There has been a great deal of research in static code analysis over the past several years, but only a small fraction of it has focussed on automated reusable part identification. Chen, Nishimoto, and Ramamoorthy briefly discuss the idea of subsystem extraction by using code information stored in a relational database [CNR90]. They also describe a tool called the C Information Abstraction System to support this process.

Caldiera and Basili [CaB91] describe a tool called Care, that helps identify reusable components according to a set of "reusability attributes" based on software metrics. These attributes include measurements of how useful the component is in its problem domain, how much it would cost to reuse it, and its quality. The idea behind Care is that it will do the initial identification of components that have strong reusability characteristics, and then

a domain expert will do a further examination of these components to determine their appropriateness to the domain, and package them for reuse. Mayobre [May91] describes how these techniques were extended and used to help identify data communication components at Hewlett-Packard.

A project to salvage reusable components from Ada source code is described by Arnold [Arn90a, Arn90b]. He discusses a number of heuristics for locating reusable components, including:

- Counting the number of references to a particular procedure or function, identifying the most frequently referenced ones as potentially reusable.
- Identifying modules that are loosely coupled with other modules.
- Identifying high-cohesion modules.

These approaches to automation are all based on static code analysis using a traditional approach to the detection of relevant software characteristics. Our approach is to employ an expert-system technique to try to make decisions with a very high degree of fidelity by identifying the design rules that are known to be supportive of reuse. In this paper, an overview of the techniques used by the toolset are presented together with the results of a preliminary assessment experiment in which its results were examined by human experts.

2. Expert System Overview

The traditional method of operation of an expert system is to mimic the behavior of a human expert in a very narrow area. The human knowledge in the application area is first captured by various means and then codified as a set of inference rules that are used to guide the decision making of an inference engine.

A human expert who is looking at existing software seeking reusable parts calls upon many forms of knowledge but the most important are (a) knowledge of the domain from which the system being examined came and (b) knowledge of the domains or subdomains that might make use of the reusable components that are recovered. The domain knowledge manifests itself as awareness of the significance of the identifiers used, the meaning of abbreviations, the importance of the fundamental design of the system, and possible future products in the domain.

The human uses this knowledge in a very complex way to guide the search for components worth salvaging. As an example, consider the actions of a human expert reviewing an existing system upon locating a module called `rt_scheduler`. With no additional information, knowledge of the abbreviation (`rt`) and the importance of the scheduling function alerts the human to the significance of the module and suggests that the module be examined in detail. Correspondingly, a module named `drv_obsolete_rdr` is unlikely to warrant further consideration.

Codifying specific domain knowledge is an approach that might be considered as the basis for a toolset of the form described here, but the knowledge required cannot be codified as a typical set of inference rules. The knowledge used in the example above is not

particularly specialized. It is, however, part of an enormous body of diverse knowledge that is called upon by the human code reviewer, and it is difficult to characterize. Rather than attempting to capture this knowledge base, we use knowledge that would typically be employed by the *original developers* of systems as the primary knowledge base for the toolset. This knowledge is restricted to design knowledge at all levels (high level, low level, and implementation) that is related to reusability, and supplemented by metrics that characterize potentially reusable parts.

As an example, it is reasonable to assume that the developer uses techniques such as functional and data abstraction to facilitate change and simplify the design when building a system. If this is the case, the resulting abstractions constitute candidates for initial selection as reusable parts. Functional abstractions that are used "frequently" by an application, for example, might be sufficiently useful that they can be reused. Similarly, abstract data types encapsulate design decisions and provide substantial functionality that might be reused.

In addition to rules defining candidate reusable structures, our model includes a secondary set of rules that is derived from the application domain (and are therefore

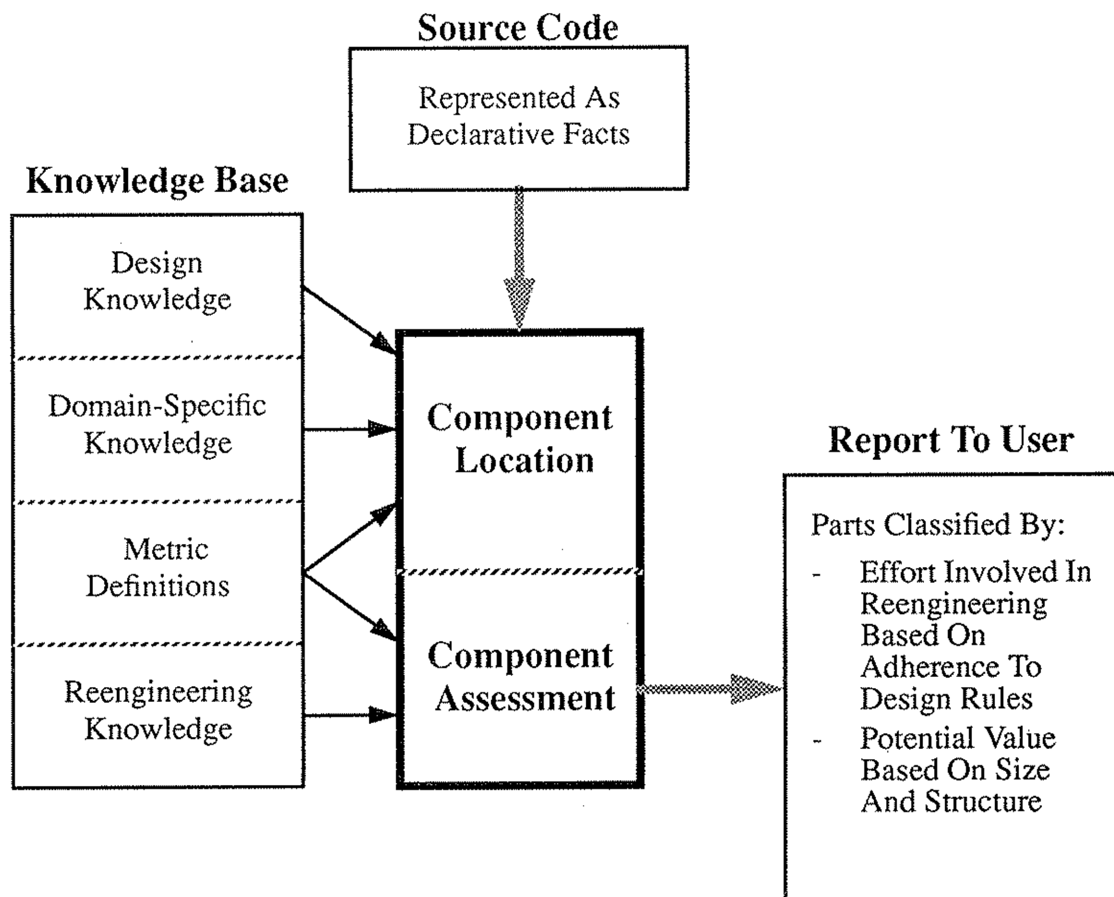


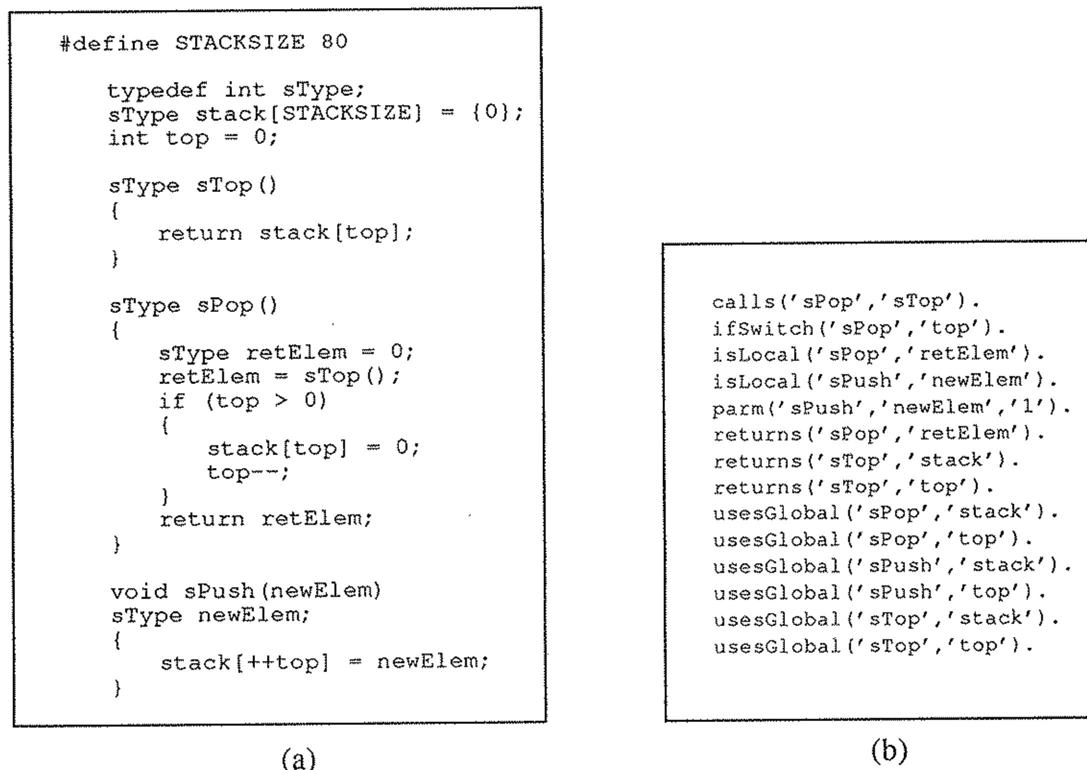
Figure 1 - Areas Of Knowledge And Their Uses

changed as needed) and a secondary set of rules that captures important stylistic characteristics that influence reengineering. The purpose of the first set is to enhance the location process so as to locate parts likely to be relevant to a spectrum of systems across the problem domain. The purpose of the second set is to enhance the selection process by assessing the ease with which a candidate part can be made ready for entry into a reuse library.

3. Toolset Overview

Code Miner is designed to assist the programmer in identifying parts of existing systems that might be potential candidates for a reuse library. The programmer can then view these candidates and pass final judgement on whether the candidates are truly useful and worth salvaging. Although our expert-system model includes the four forms of knowledge shown in Figure 1, the prototype system implements an experimental design knowledge base and a simple reengineering knowledge base. These were selected for first implementation because they represent the most significant technical challenges

All of the knowledge employed by the system is codified in Prolog including an extensive set of facts about the software being examined. The system consists of three major functional elements:



```
#define STACKSIZE 80

    typedef int sType;
    sType stack[STACKSIZE] = {0};
    int top = 0;

    sType sTop()
    {
        return stack[top];
    }

    sType sPop()
    {
        sType retElem = 0;
        retElem = sTop();
        if (top > 0)
        {
            stack[top] = 0;
            top--;
        }
        return retElem;
    }

    void sPush(newElem)
    sType newElem;
    {
        stack[++top] = newElem;
    }
```

(a)

```
calls('sPop','sTop').
ifSwitch('sPop','top').
isLocal('sPop','retElem').
isLocal('sPush','newElem').
parm('sPush','newElem','1').
returns('sPop','retElem').
returns('sTop','stack').
returns('sTop','top').
usesGlobal('sPop','stack').
usesGlobal('sPop','top').
usesGlobal('sPush','stack').
usesGlobal('sPush','top').
usesGlobal('sTop','stack').
usesGlobal('sTop','top').
```

(b)

Figure 2 - Stack Implementation And Corresponding Prolog Facts

- A *C parser* that generates abstract syntax trees from C source text.
- A *Prolog interpreter.*
- An *interactive front end.*

The *C parser* reads the program, performs the normal C preprocessing steps, and builds an abstract syntax tree of the entire program in main memory. The tree is then searched for a number of syntactic patterns. When a matching pattern is found, information about the pattern is captured in the form of a Prolog fact and written to a file. For example, all function invocations, along with invoking functions, are found and saved in the form `calls(CallingFunc,CalledFunc)`. Figure 2 (a) shows an example of the input C code for a small stack implementation, and Figure 2 (b) the corresponding Prolog fact output.

The *Prolog interpreter* provides the inference mechanism by which candidate reusable components are identified. The interpreter is invoked dynamically upon user request. It acquires information about the C program to be analyzed from the set of facts written by the parser, and then applies the sets of Prolog rules codifying the various forms of knowledge to these facts, reporting on all source-code parts that satisfy these rules. This process is explained in more detail in the next section.

The *interactive front end* allows the user to select a program to be analyzed, invokes the Prolog interpreter, and sends the results of the analysis back to the user in a separate display. Figure 3 shows how these system elements fit together.

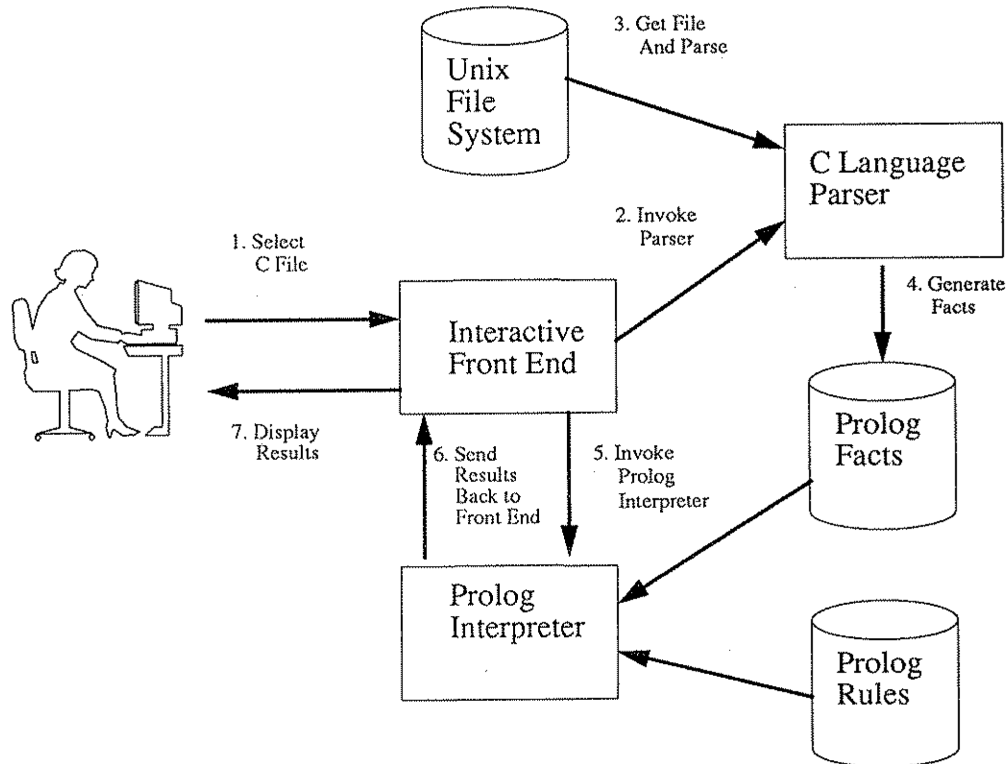


Figure 3 - Code Miner Processing Sequence

4. Knowledge Base

Although four forms of knowledge are shown in Figure 1, the focus in the prototype system is on the design knowledge. In its current form, the design knowledge permits the system to identify potentially reusable parts in three ways:

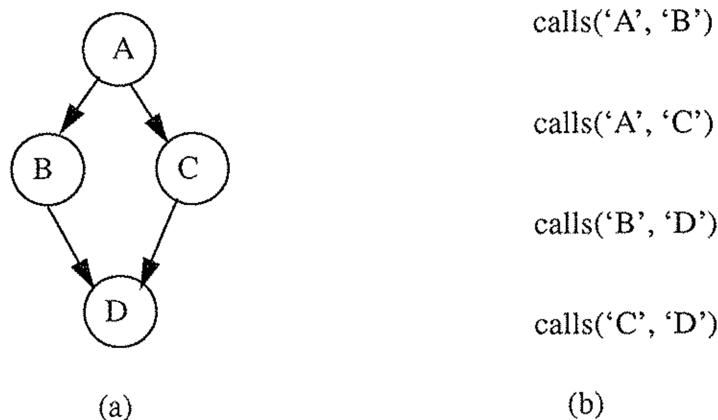
- By identifying functions that are invoked multiple times from multiple sections of code.
- By identifying functions that are loosely coupled.
- By identifying functions and global data elements that can be grouped together to form abstract data types.

Thus, only the structural relationships between major syntactic units are examined. Simpler aspects of the code relevant to reuse, such as functional complexity, have not considered in the current implementation.

4.1 Multiply-Invoked Functions

To determine the first item mentioned above, the only information needed is the set of invoked and invoking functions and the associated call graph. So, for example, if function A invokes functions B and C, and functions B and C invoke function D, the graph would appear as shown in Figure 4 (a). A set of Prolog facts modelling this graph, shown in Figure 4 (b), would be generated during the program analysis phase.

It is a simple matter to traverse the Prolog interpretation of this graph, saving those nodes that have more than one edge leading into them. Graph traversal algorithms specified in Prolog can be found in Clocksin and Mellish [ClM87] and Sterling and Shapiro [StS86]. In this simple example, function D could be flagged as potentially reusable because of its multiple invocations. To avoid flagging the standard C library functions as reusable, a relation is included in the set of Prolog rules that identifies all of these functions, and any that are invoked are ignored.



calls('A', 'B')

calls('A', 'C')

calls('B', 'D')

calls('C', 'D')

(b)

Figure 4 - Program Call Graph And Its Prolog Translation

4.2 Coupling

Informally, *coupling* refers to how tightly or loosely bound a set of modules are to each other. Functions that are loosely bound tend to be easier to remove and use in other contexts than those that depend heavily on other functions or non-local data. Determining coupling information is complex because there are several different types of coupling, and the concepts of *looseness* and *tightness* are difficult to quantify.

The types of coupling checked for in Code Miner are:

- *Data coupling*: The functions share simple data through their interfaces.
- *Common coupling*: The functions share global data.
- *External coupling*: The functions share data with the outside world.
- *Control coupling*: The functions share data items upon which control decisions (e.g., branching) are made.

Thus, in order to determine coupling characteristics, the knowledge base has to be aware of not only the program's call structure, but also which global data elements are shared by which functions, which data elements are passed from function to function as parameters and returned values, whether parameter values are being used within a function to make branching decisions, and whether a function invokes input or output functions.

Once again, this information is captured in a set of Prolog facts by the C parser. Some important facts in this context are:

- *calls(F, G)* - indicates that function F calls function G.
- *usesGlobal(F, D)* - indicates that function F uses global data D.
- *ifSwitch(F, D)* - indicates that data name D is used by function F to make a branch decision.
- *returns(F, D)* - indicates that function F returns data element D.
- *parm(F, D, N)* - indicates that data name D is the Nth parameter to function F.

As an example of the use of these facts, consider the case of control coupling. If function A calls function B, and function B takes data X as a parameter, and data X appears in a boolean expression inside an if-statement, then functions A and B are control coupled. This is captured by the Prolog rule:

```
controlCoupled(F,G) :- calls(F,G), parm(G,X,N), ifSwitch(G,X),
F\==G.
```

Currently the system assumes that if none of the above coupling characteristics apply to a function, then that function is reusable. More work is needed to determine how this restriction can be relaxed so that candidates with varying degrees of coupling can be identified and flagged as possibly reusable.

4.3 Abstract Data Types

A set of functions that use the same global data elements can often be regarded collectively as an abstract data type. An example of where it would be useful to identify

such a cluster of functions and data is when a C program is being converted into a C++ program, and the programmer is creating a set of appropriate C++ classes.

To identify potential abstract data types, the program is regarded as a bipartite directed graph, where nodes are either function names or global data names, and edges are directed from function nodes to data nodes and specify a "uses" relationship. Figure 5 shows an example of a program fragment implementing a simple queue, and how this would be thought of as a bipartite graph.The algorithm operates by performing a depth-first traversal of the graph looking for strongly connected components. Each strongly connected component is reported to the user as potential abstract data type. Because this type of search problem is not linear in the number of nodes, the search depth has to be limited. However, early results indicate that in spite of the search limit, a substantial number of potential ADTs

```
#include <stdio.h>
#define QLIMIT 10

typedef int qType;

qType queue[QLIMIT] = {0};

int front = 0;
int back = 0;
int entries = 0;

void enqueue(newQElem)
    qType newQElem;
{
    if ((back <= QLIMIT-1) &&
                (entries <= QLIMIT))
    {
        queue[back++] = newQElem;
        entries++;
    }
    else if ((back > QLIMIT-1) &&
                (entries <= QLIMIT))
    {
        back = 0;
        queue[back++] = newQElem;
        entries++;
    }
    else if (entries > QLIMIT)
    printf ("Queue full");
}

qType dequeue()
{
    if (entries == 0)
    {
        printf("%s\n", "Queue empty");
        return -1;
    }
    else if (front > QLIMIT - 1)
    {
        front = 0;
        entries--;
        return queue[front++];
    }
    else
    {
        entries--;
        return queue[front++];
    }
}
```

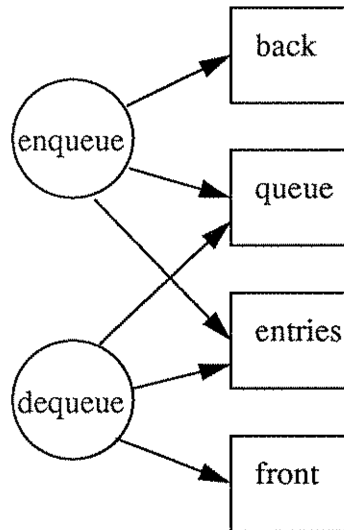


Figure 5 - Possible ADT And Its Bipartite Graph

can be located, even in large program files.

5. Empirical Results

To evaluate the usefulness of this approach, we have conducted a preliminary experiment in which the toolset was applied to some large code sections and its performance rated by human experts. The evaluation was done on five public-domain systems, all of which were products of the Free Software Foundation, Inc., with a total executable source-line count of about 48,500. Table 1 shows details of the systems in question, along with a count of how many individual reusable functions were identified.

Table 1:

System	Size In Executable Statements	Approximate Number of Functions	Possibly Reusable Functions Identified
gawk 2.13	11,741	383	93
gnuchess 4.0	10,687	226	82
gnuplot 3.2	16,449	312	191
grep 1.6	4,475	122	18
sed 1.09	5,228	142	36

The evaluators participating in this experiment were eight graduate students in either Computer Science or Electrical Engineering, with experience levels in C ranging from 1 year to 8 years, with an average of 3.7 years.

The tool was run on each of these systems, and a set of reports were generated. These reports were distributed to the evaluators, and each was asked to rate subsets of the parts identified according to three criteria:

- *Practicality*: How useful would this part be, either in another system in this problem domain, or in systems in other domains?
- *Reusability*: How much effort would it take to reengineer this part in order for it to be a reasonable candidate for a reuse library?
- *Understandability*: How difficult is it to get a basic idea of what this part does?

Each of the criteria was scaled from 1 to 5, with 1 denoting an undesirable quality (not at all practical, requires extensive reengineering, or extremely difficult to understand), and 5 denoting a desirable quality (can be useful in many different contexts, can be reused with little or no reengineering, or very easy to understand).

In all, about 230 artifacts were evaluated of which 34 were abstract data types and the rest functions. This disparity between the number of functions in the systems and the

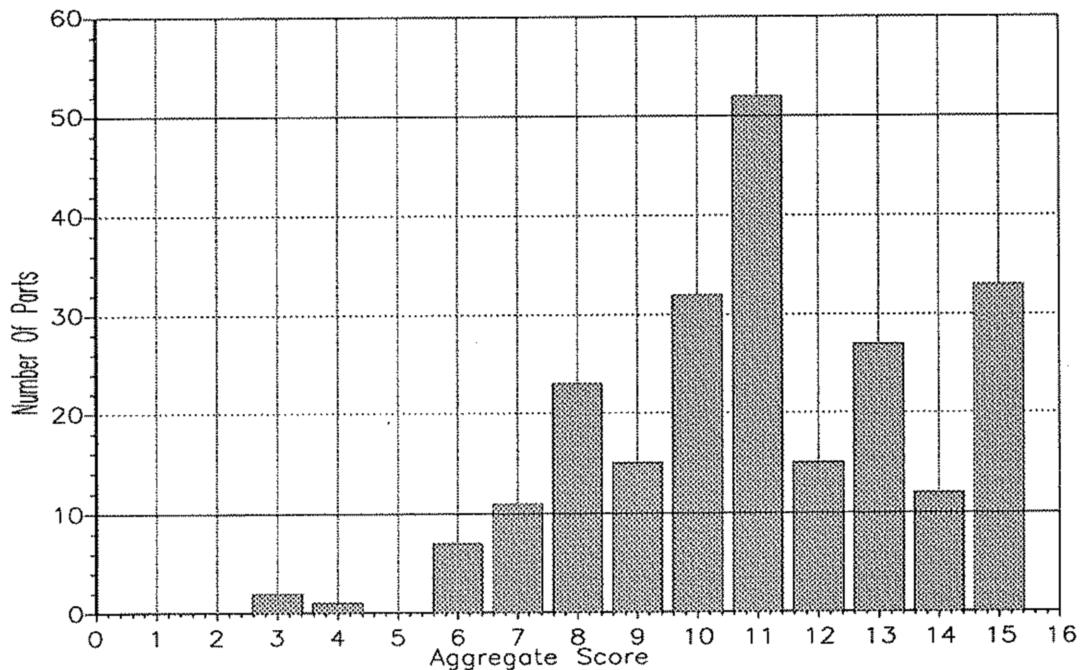


Figure 6 - Aggregate Reusability Results

number of functions evaluated is due primarily to time constraints. The average amount of time spent by each evaluator was about 2.3 hours. This is enough time to get a rough idea of the reusability of a useful subset of the components, but not enough for a detailed analysis of the entire set of systems.

The ratings assigned to each of the criteria for the 230 parts were summed to provide an aggregate reusability score, ranging from 3 to 15. Figure 6 shows a histogram of these results. We note that more than half of the components identified by the toolset were rated with a cumulative score of 10 or higher by the human evaluators. We feel that this is a positive indication of the toolset's effectiveness.

Figures 7, 8, and 9 give a more detailed breakdown of these results, according to the practicality, reusability, and understandability scores. While the results for the reusability and understandability constitute approximately normal distributions, the trend suggested in the practicality results is rather striking. We hypothesize that because the non-ADT artifacts selected by the tool are loosely-coupled and multiply-invoked functions, one can regard them as potential general-purpose functional abstractions. This hypothesis is supported by examining the functionality of many of these artifacts. The tool has a distinct bias toward selecting non-domain-specific functions intended for such areas as string and buffer manipulation, math, error processing, and graphics.

6. Conclusions

While it is too early to make any definitive statements about effectiveness of the

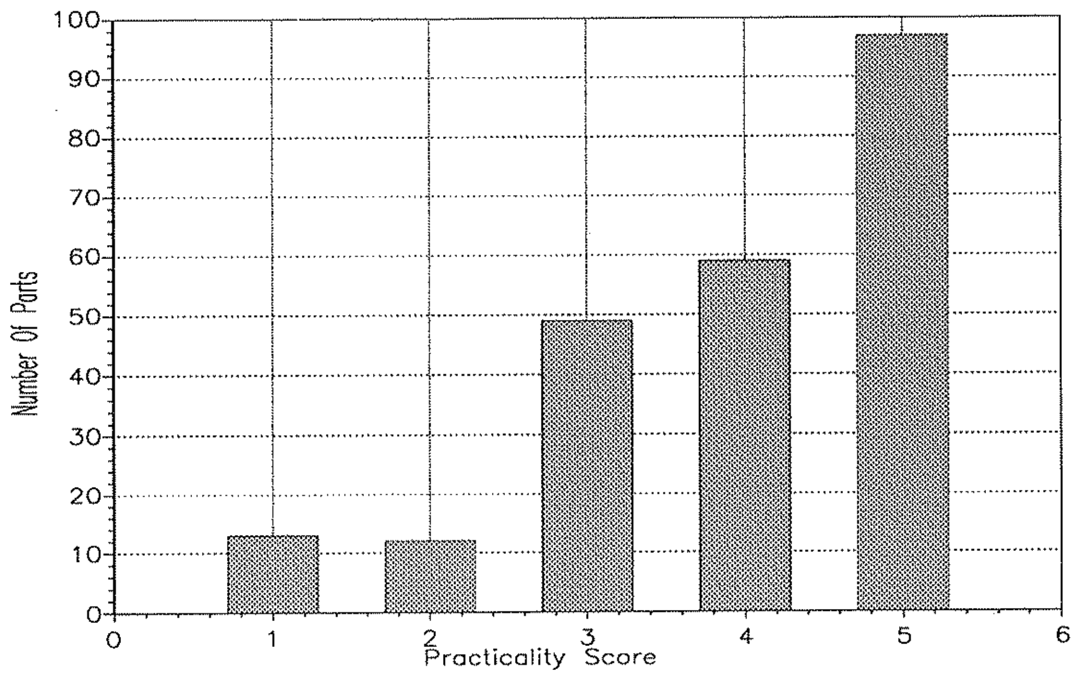


Figure 7 - Cumulative Assessment Of Practicality

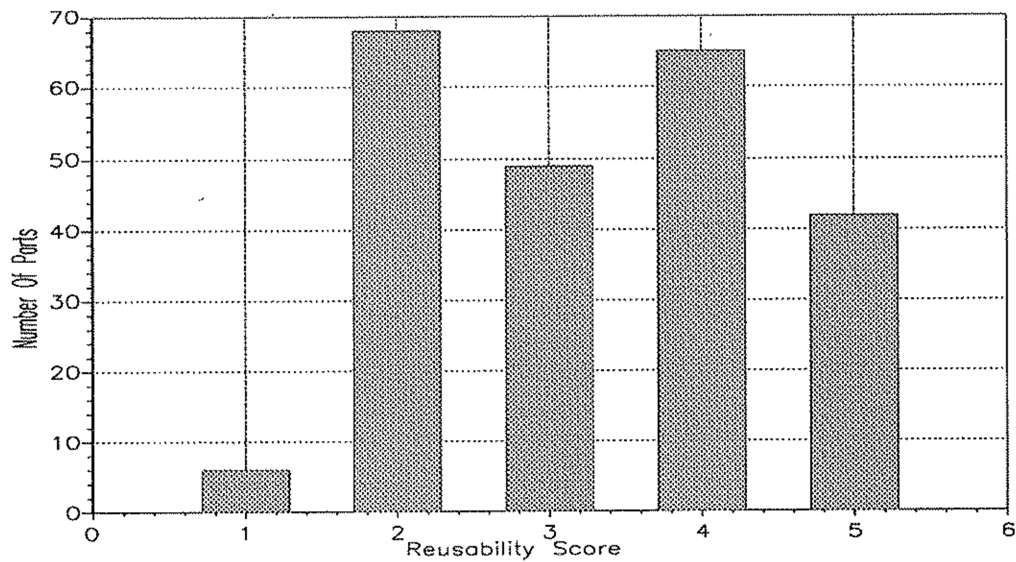


Figure 8 - Cumulative Assessment Of Reusability

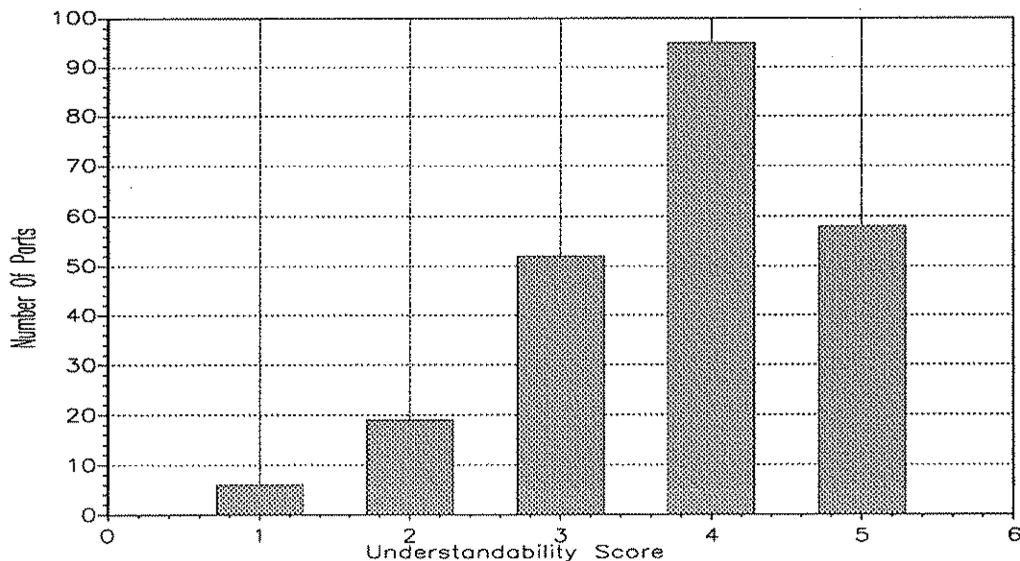


Figure 9 - Cumulative Assessment Of Understandability

scavenging method described in this paper, the results outlined in the previous section suggest that the ideas implemented in the tool merit further study. Most importantly, it is clear that a great deal of useful information about a system can be gathered by examining its structure. While a number of the evaluators raised concerns about the tool's effectiveness in discerning reasonable abstract data types, it is encouraging that a number of the ADTs identified received high aggregate scores.

It is equally clear that human judgement will remain a crucial element in the scavenging process for the foreseeable future. While the tool is capable of discerning the structure of a piece of code, it takes an experienced programmer to decide the potential usefulness of any given candidate component. Research questions remain in the areas of additional selection rules to make the tool a more discerning judge of reusability, and development of the user interface so that a human using the tool can adequately evaluate a large amount of code in a small amount of time. These two topics will be the primary focus of our future work in this area.

7. Acknowledgments

We gratefully acknowledge the participation of the following people in the preliminary evaluation of the Code Miner tool: William Dixon, Darrell Kienzle, Chris Koeritz, Doug Lamb, Will McClennan, Fraser Street, Kevin Wika, and Dallas Wrege. This work was funded in part by the Virginia Center for Innovative Technology grant number INF-92-001.

References

[Arn90a] Arnold, R.S., *Heuristics For Salvaging Reusable Parts From Ada Source Code*, SPC Technical Report, ADA_REUSE_HEURISTICS-90011-N, March 1990.

[Arn90b] Arnold, R.S., *Salvaging Reusable Components From Ada Source Code: A Progress Report*, SPC Technical Report, SALVAGE_ADA_PARTS_PR-90048-N, September 1990.

[CaB91] Caldiera, G. and V.R. Basili "Identifying and Qualifying Reusable Software Components", *IEEE Computer*, February 1991.

[ClM87] Clocksin, W.F. and C.S. Mellish. *Programming in Prolog*, Springer-Verlag, 1987.

[CNR90] Chen, Y-F, M.Y. Nishimoto, and C.V. Ramamoorthy "The C Information Abstraction System", *IEEE Transactions on Software Engineering*, Vol. 16, No. 3, March 1990.

[DuK91] Dunn, M.F. and J.C. Knight, "Software Reuse in an Industrial Setting: A Case Study", *Proceedings of the Thirteenth International Conference On Software Engineering*, Austin, TX, 1991.

[May91] Mayobre, G. "Using Code Reusability Analysis to Identify Reusable Components from the Software Related to an Application Domain", *Proceedings of the Fourth Annual Workshop On Software Reuse*, Reston, VA, November, 1991.

[Sel88] Selby, R.W. "Empirically Analyzing Software Reuse in a Production Environment", in *Software Reuse: Emerging Technology*, W. Tracz, ed., IEEE Computer Society Press, 1988.

[StS86] Sterling, L. and E. Shapiro, *The Art of Prolog*, The MIT Press, 1986.