Identifying Enumeration Types Modeled with Symbolic Constants

John M. Gravley and Arun Lakhotia
Center for Advanced Computer Studies
University of Southwestern Louisiana
Lafayette, LA 70504
{ jmg, arun }@cacs.usl.edu

Published in: Proc. of the 3rd Working Conference on Reverse Engineering, © IEEE CS Press, November 1996.

Abstract

An important challenge in software reengineering is to encapsulate collections of related data that, due to the absence of appropriate constructs for encapsulation in legacy programming languages, may be distributed throughout the code. The encapsulation of such collections is a necessary step for reengineering a legacy system into an object-oriented design or implementation. Encapsulating a set of related symbolic constants into an enumeration type is an instance of this problem. We present a classification of how enumeration types are modeled using symbolic constants in real-world programs, a set of heuristics to identify candidate enumeration types, and an experimental evaluation of these heuristics.

1 Introduction

There is a growing interest in the problem of finding objects and abstract data types in legacy code [1, 2, 4, 10, 11, 14, 23, 25]. Most researchers have presented solutions which assume that related data items (belonging to an object) are already encapsulated using constructs such as structs, common blocks, records, types, etc. This paper addresses the problem of grouping related data items that were not previously encapsulated. Our research thus attempts to fill the void left by other researchers.

Identifying enumeration types modeled with symbolic constants is an instance of the larger problem of encapsulating related data and is conceptually simpler than identifying the elements of a record type. The problem is sufficiently rich at deeper levels to expose all of the issues involved in the larger problem. Hence, this problem is well-suited for comprehensive investigation in a laboratory environment and will provide considerable insight into solving the larger problem.

Our first step in solving this problem is a comprehensive analysis of how enumeration types are modeled in real-world C programs that have evolved through various dialects of the language. This analysis, carried out by studying several real-world programs, has yielded a classification that may be used to describe the problem space. This classification is analogous to the terminology developed by Rugaber et al. [21] for describing issues pertinent to the problem of understanding interleaved code.

Creating collections of symbolic constants such that each collection constitutes an enumeration type may be viewed as a classification problem. Numerous techniques have been proposed for classifying various elements of software systems for varying purposes [9]. Some of these techniques discover the modular architecture of undocumented legacy code [11, 13, 14], others group program components that are related to the same error [7], and still others simply impose a structure upon an otherwise large, complex, and unstructured data [3, 12]. However, there is no single classification technique that is applicable to all problems. For a new problem, either an existing technique is customized or a new technique that uses special properties of the problem domain is developed.

Our heuristics for grouping macro symbols are based upon the principle that: Symbols that are *defined* (or *declared*) in the same context are placed in the same group. This is a significant departure from the principle upon which previous subsystem classification techniques are based, i.e., Symbols that are *used* in the same context are placed in the same group. In the context of recovering enumeration types, our principle leads to techniques that are computationally inexpensive and yet do not incur a significant loss in precision.

We performed an experiment to evaluate the effectiveness of our heuristics in creating candidate enumeration types that correspond to actual enumeration types. We

This work was partially sponsored by the Department of the Army through the Army Research Office. However, the contents of this paper do not necessarily reflect the position or the policy of the government, and no official endorsement should be inferred.

enum-specifier ::=	enum identifier$_{opt}$ { enum-list }
enum-list ::=	enumerator
	| enum-list, enumerator
enumerator ::=	identifier
	| identifier = constant-expression

Figure 1. Syntax of the enumeration construct (enum) of ANSI C and C++. [8, 24].

```
enum Day { SUN, MON, TUE, WED,
   THU, FRI, SAT };
```

Figure 2. Example in which the enumerator values are implicitly defined by the compiler.

```
enum Opcode { ADD = 10,
   SUB = 32, MUL = 25 };
```

Figure 3. Example in which the enumerator values are explicitly defined by the programmer.

```
enum {            typedef enum {        enum Color {
   RED = 1,          RED = 1,              RED = 1,
   GREEN,            GREEN,                GREEN,
   BLUE };           BLUE } Color;         BLUE };
      (a)                 (b)                  (c)
```

Figure 4. Examples of how enumerations may be declared explicitly (a) using the enum construct, or (b) using the enum construct in conjunction with a typedef. Example (a) is an *anonymous* enumeration.

compared the candidate enumeration types recovered by our techniques with the *ideal* set of enumeration types as defined by an oracle—in our case, a programmer [9]. The data from evaluating our heuristics on eight moderate-sized, real-world software systems suggests that heuristics that classify symbols based upon the proximity of the declarations can give good results for discovering candidate enumeration types in preprocessor symbols.

The rest of this paper is organized into the following sections. We present an overview of C and C++ enumeration types in Section 2. In Section 3, we describe the various ways that enumerations are modeled using symbolic constants in C and C++ programs. We present our heuristics for identifying candidate enumeration types in Section 4. We present the design of our experiment in Section 5 and analyze the results in Section 6. Finally, we present our conclusions in Section 7.

2 Enumeration types

"An enumeration is an ordered list of [symbolic] values" [16, p. 137] typically used to introduce a finite (usually small) number of symbolic values. This section gives an overview of C and C++ enumerations. It introduces the salient issues pertaining to constructing, manipulating, and coercing the values of enumeration types in the context of any programming language with emphasis on ANSI C and C++ [24]. The section also introduces operations that may be considered violations in C++ and highlights operations that C++ permits but may be violations in other languages, such as Ada.

Figure 1 shows the syntax for ANSI C or C++ enumeration types. The two components of interest in an enumeration are a set of literal symbols called the *enumerators* (`enum-list`) and the *name* (identifier$_{opt}$) of the enumeration. The enumeration name introduces a new type whose values consist of the set of enumerators. An enumeration without a name is called an *anonymous* enumeration.

The identifiers (or symbols) in the `enum-list` are considered constants and may be used as such. Their values may either be defined implicitly by the compiler (Figure 2) or be explicitly provided as a `constant-expression` (Figure 3). The symbols in an `enum-list` are implicitly equated to successive integers starting from zero. The assignment of a `constant-expression` forces the corresponding symbol to be equated with the value of that expression and forces the values of the successive symbols to be equated with the successive integer values. Figure 4 gives examples of how the colors RED, GREEN, and BLUE may be explicitly defined as an enumeration type in some versions of C and in C++.

An `enum` declaration may have a combination of the explicit and implicit mappings, with some enumerators mapped to explicit values and others mapped to implicit values, as in:

```
enum MyBool { NO, YES, OFF = 0, ON };
```

Such an explicit mapping of enumerators to integers has the side effect that two enumerators in the same enumeration may be mapped to the same integer. In the above example, both `NO` and `OFF` are mapped to zero, and `YES` and `ON` are mapped to one. This contrasts C and C++ enumerations with those in Ada, where each enumerator (in an enumeration) is mapped to a unique integer. Thus, unlike Ada enumerations, C and C++ enumerations are *not* ordinal types.

We refer to enumerators with the same integer mapping within the same enumeration as *aliases*. This word is commonly used to group symbols that refer to the same memory location. However, since the literal symbols are constants which do not have memory locations, we extend the usage of this word to two or more symbols mapped to the same integer in the same context.

An enumerator that belongs to at least two different enumerations is said to have *multiple identity*. A language must provide a way to statically resolve the name space conflict in determining which enumeration an enumerator belongs to in order to permit enumerators with multiple identity.

Values of an enumeration type are created by the literal constants and may be assigned to a variable of that type, as in:

$$\begin{array}{rl} \mathit{Literal\ Symbol}: & \rightarrow \mathit{Enum} \\ \mathit{Relational\ Op}: & \mathit{Enum} \times \mathit{Enum} \rightarrow \mathit{Boolean} \\ \mathit{CoerceInt}: & \mathit{Enum} \rightarrow \mathit{int} \\ \mathit{CastInt}: & \mathit{int} \rightarrow \mathit{Enum} \\ =: & \mathit{Enum} \times \mathit{Enum} \rightarrow \mathit{Enum} \end{array}$$

Figure 5. Signatures of operations that may be performed upon C++ enumerations.

```
Day StartDay; StartDay = MON;
```

Languages that enforce strict type checking do not permit assignment between two different enumeration types, which makes

```
StartDay = RED;
```

incorrect in such languages. Similarly, the assignment of integer values to an enumeration variable:

```
StartDay = 2;
```

may not be permitted if strict type checking is enforced.

A language may, however, allow coercion of an enumerator to an integer when it is assigned to an integer variable or is part of an integer expression, e.g.,

```
int i; i = RED;
```

This is useful when some computation is performed on all of the values of an enumeration, as shown in the following C++ fragment:

```
Day d;
for (d = SUN; d <= SAT; d = Day(d + 1))
    ; // do whatever
```

The expression `Day(d + 1)` in this example explicitly casts the value of expression `d + 1` to be of type `Day`. The above loop has the unfortunate effect that it terminates with an out of range value of `d` (i.e., it computes `SAT + 1`, which is not a member of `Day`). However, the cast is necessary because the statement `d = d + 1` would be incorrect since `d + 1` is implicitly coerced to an integer.

A loop such as that shown above may be used to enumerate all of the values of an ordered enumeration when its symbols are mapped to integer values with a known, periodic interval, even if the period is not one. It may become cumbersome with enumerations, such as those in Figure 3, whose numeric values have been explicitly defined and do not follow any regular interval.

The operations that may be performed on an enumeration type, say `Enum`, are summarized by the signatures in Figure 5. The last signature is for assignment and has three terms because assignment is an expression in C++.

3 Modeling enumeration types using symbolic constants

This section discusses the modeling of enumeration types in real-world C programs. This modeling may be studied based upon (a) how the set of enumerators of an enumeration is declared, (b) how its name is declared, and (c) how the association between the set of enumerators and the name is defined. Figure 6 gives two ways of modeling enumerations equivalent to those in Figure 4.

```
/* define colors */
#define RED 1
#define GREEN 2
#define BLUE 3
/*
* end of color
*/
```
(a)

```
typedef int Color;
const RED = 1;
const GREEN = 2;
const BLUE = 3;
```
(b)

Figure 6. Examples of how enumerators may be declared implicitly (a) using the `#define` directive, or (b) using the `const` construct in conjunction with a `typedef` to declare an enumeration name.

```
/* some code */
#define RED 0
/* some more code */
#define GREEN 1
#define BLUE 2
```

Figure 7. Example of a *distributed* enumerator declaration. This C code fragment shows a logical group of symbols that form an enumeration even though they are interleaved with other code. The interleaving may even occur across files.

3.1 Declaration of enumerators

The enumerators belonging to an enumeration may be declared using any combination of:

1. `#define` directives,
2. `const` constructs,
3. an `enum` construct,

All of the literal symbols belonging to an enumeration are *encapsulated* when declared using an `enum` construct (Figure 4) but are *unencapsulated* if the other constructs are used. The enumerators in the first two cases above are frequently grouped by declaring them in succession and bounding the declarations by visual cues, as shown in Figure 6(a). We call such a declaration *contiguous*. The enumerators may be interleaved [21] with other declarations, or even across several files when the declarations are unencapsulated, as illustrated in Figure 7. We refer to such declarations as *distributed*.

A more complex *intertwining* of enumeration declarations is created due to enumerators with *aliases* or with *multiple identity*. Two such cases are illustrated by the code fragments in Figures 8 and 9.

The fragment of Figure 8 may be interpreted in two ways. The first interpretation is that it introduces two enumerations, one introduced by the set of `#define` directives and the other by the `enum` construct. The two enumerations are in some sense equivalent since there is a one-to-one correspondence between their enumerators and

```
/* extracted from EDGE */
#define BLACK_BOX 0
#define GREY_BOX 1
#define WHITE_BOX 2
#define SEPARATE_GRAPH 3

typedef enum { black = BLACK_BOX,
        grey = GREY_BOX,
        white = WHITE_BOX,
        separate = SEPARATE_GRAPH }
         statuses;
```

Figure 8. Example of *aliased* enumerators found in the EDGE system [15]. A similar effect could also be achieved using a combination of `#define` directives and `const` constructs to introduce the enumerators.

```
typedef int Starch;
#define Corn 0
#define Rice 1
#define Potato 2
#define Bean 3

typedef int Grain;
/* #define Corn 0 */
#define Wheat 1
#define Rye 2
#define Barley 3
#define Sorghum 4
```

Figure 9. Example of an enumerator with *multiple identity*. The duplicate symbol `Corn` is commented out just to highlight that it also belongs to the latter enumeration. In actual code such a comment may not necessarily exist [5].

the corresponding enumerators have the same integer mapping. The second interpretation is that the code fragment introduces one enumeration consisting of the union of symbols introduced by the `#define` directives and the `enum` construct. The enumerators consist of pairs of aliases. Irrespective of which interpretation is chosen, it is clear that a programmer may interchangeably use the corresponding pairs of enumerators in the two sets.

Figure 9 illustrates intertwining caused by using an enumerator with multiple identity. Creating an enumerator with multiple identity using symbolic constants requires that the enumerator have the same value in all of the enumerations. The enumerator `Corn` belongs to both enumerations in this case and is mapped to the value 0. However, in the program text this enumerator may be declared only once. Thus, it may not actually appear in the second list, not even as a comment.

3.2 Declaration of enumeration names

An enumeration may have a type name whether or not its

- **Enumerators**
  - Declaration
    - encapsulated
    - unencapsulated
      - contiguous
      - distributed
  - Intertwined
    - aliases
    - multiple identities
- **Enumeration name**
  - Declaration
    - exists (named)
      - using a `typedef` name
      - using an `enum` tag name
    - does not exist (anonymous)
  - Association with enumerators
    - attached
    - contiguous
    - distributed

Figure 10. A classification of the various issues in the declaration of enumerations in C programs.

enumerators are encapsulated. However, the name can be directly associated, i.e., *attached*, to the enumerators only when they are encapsulated since only the `enum` construct permits such an association. Names associated with a set of unencapsulated enumerators, or even encapsulated enumerators, may be introduced by using a type declaration or a macro declaration, as in:

```
typedef int Day;
#define Color int
```

Since the association between unencapsulated enumerators and their type name is not syntactically explicit, the association is usually provided by visual cues such as placing the two declarations in physical proximity. Such an association is termed as *contiguous*. This association may not be available, even by visual cues, if certain coding standards require that all type declarations be grouped in one location, all macro definitions be in another location, and all constants be in yet another location. Such an association is termed as *distributed*. Finally, an enumeration that does not have any type name associated to it is termed an *anonymous* enumeration. In summary, the enumeration declarations in C programs may be classified as shown in Figure 10.

4 Identifying candidate enumeration types

This section presents our heuristics for identifying candidate enumeration types by grouping macro symbols introduced using `#define` directives. The heuristics are "derived" from the classification presented in Section 3 (summarized in Figure 10).

4.1 Problem formulation

Identifying candidate enumeration types that have been created using macro symbols requires partitioning the macro symbols of a program into sets of symbols, where each set logically constitutes an enumeration type. Our heuristics for identifying candidate enumeration types are based upon the principle that symbols that are *defined* (or *declared*) in the same context are placed in the same group. This principle is based upon the observations that:

1. macro symbols representing an enumeration type are most often placed in the same file,
2. these symbols are most often declared in a lexical sequence.

The above observations further imply: If related macro symbols are (most often) declared in the same file in a lexical sequence, then the problem of identifying candidate enumeration types is to split the sequence of `#define` directives in a file into non-empty, non-overlapping subsequences such that the macro symbols introduced in each subsequence represent a candidate enumeration type.

4.2 Problem space

The `#define` directive provides a mechanism to define symbolic macros that are expanded, often referred to as preprocessed, before a C program is compiled. This directive is used to introduce symbolic names for constants, to introduce macro "functions," and to introduce constructs that are not directly supported by the language. Furthermore, the directive may be used to introduce *expressions* that evaluate to constants after preprocessing.

The only macro symbols that are candidates for becoming enumerators are those that have a value in the range of type `int` when expanded since symbols introduced by `enum` constructs are mapped to values of type `int`. Hence, all other macro symbols may be excluded from the problem space.

4.3 Heuristics

This section present nine heuristics for grouping `#define` directives to create candidate enumerations. The first six heuristics are the subjects of experimental evaluation.

The first four heuristics are based upon the intuition that programmers leave non-syntactic "visual cues" to delineate blocks of related code.

1. *Blank Line* heuristic: Group successive `#define` directives bounded by lines consisting of "white space."
2. *Line Comment* heuristic: Group successive `#define` directives bounded by lines consisting of a complete comment (i.e., `/* ... */`).
3. *Block Comment* heuristic: Group successive `#define` directives bounded by lines consisting of a complete comment across two or more lines.
4. *Any Visual Cue* heuristic: Group successive `#define` directives bounded by lines consisting of any of heuristics 1–3.

The next two heuristics are based upon the intuition that since enumeration types are ordinal types in many languages, it is likely that programmers assign consecutive numeric values to the symbols in an implicit enumeration. These heuristics group consecutive `#define` directives whose values are either increasing by one or decreasing by one continuously. For example, if a consecutive sequence of macro symbols introduces the values 0, 1, 2, 3, 10, then the symbols introducing 0, 1, 2, and 3 are placed in the same group.

5. *Sequence Level I* heuristic: Group all successive `#define` directives whose values form either an increasing or a decreasing sequence.
6. *Sequence Level II* heuristic: Create groups based upon *Any Visual Cue* heuristic and decompose each group into groups using the *Sequence Level I* heuristic.

The next heuristic provides the minimum baseline for evaluation in that it represents the worst case grouping of `#define` directives in an include file. This heuristic is used as a *control* as part of the evaluation of heuristics 1–6.

7. *File* heuristic: Place all of the `#define` directives in a file in a group.

The last two heuristics were used by our oracle, a programmer, to manually generate the expected, or best case, groupings of `#define` directives. They were not implemented in our prototype because they are subjective, and therefore imprecise. See Section 5.3 for more discussion.

8. *Lexical Similarity* heuristic: Group `#define` directives whose symbols contain "significant" common prefixes, suffixes, or other character sequences. This heuristic is influenced by the observation that programmers tend to use common prefixes or suffixes for symbols that are related. This heuristic is intuitively very easy to understand but very difficult to define precisely.
9. *Domain Similarity* heuristic: Group `#define` directives that introduce symbols that are related based upon programming and domain knowledge.

5 Experiment design

This section presents the design of our experiment to evaluate the effectiveness of the heuristics presented in Section 4.3. It describes (a) the subject systems used for the experiment, (b) the prototype that implemented these heuristics along with its limitations, (c) the oracle used for identifying the expected sets of enumeration types, and (d) the measure of effectiveness used for analyzing the results.

5.1 Subject systems

Two important criterion have influenced our choice of subject systems for this experiment. First, the systems must be representative of the real-world. Second, they must be available to other researchers for repeating our experiment or comparing our results with the results of other heuristics. Based upon these criteria, we chose the following eight systems:

1. BTOOL: A set of tools for measuring branch coverage of C programs from the University of Illinois (Brian Marick: marick@testing.com).
2. EDGE: Extendible graph editor from the University of Karlsruhe [15].
3. FIELD: Friendly Integrated Environment for Learning and Development from Brown University [18].
4. GCC: GNU C/C++ Compiler version 2.7.1 from the Free Software Foundation.
5. GHOSTSCRIPT: PostScript interpreter version 2.6.0 from Aladdin Enterprises (PostScript is a trademark of Adobe Systems, Inc.)
6. SGEN: Synthesizer Generator from GrammaTech, Inc. [19, 20].
7. Standard Unix Include Files for SunOS Release 4.1.3_U1 (Unix is a registered trademark of AT&T).
8. WPIS: Wisconsin Program Integration System from University of Wisconsin [6].

All of these systems are real-world in that they have "lived" for several years now, have undergone multiple releases, and are being used by several people other than their developers. Of these, BTOOL, EDGE, FIELD, GCC, and Ghostscript are available in the public domain; SGEN and the Unix include files are commercial products; and WPIS, a product of academic research, is available for a modest fee.

Table 1 summarizes some of the interesting statistics of our subject systems. We have assumed that all of the (relevant) `#define` directives for these systems are available in the include files, i.e., the `.h` files. This table presents the number of these files and their sizes, the number of `#define` directives in each system, and the number of macro symbols that are enumerator *candidates*. The criteria for selecting these candidates will be discussed later.

5.2 Prototype

We developed a prototype system that implements the heuristics presented in Section 4. The following compromises were made to facilitate the quick development of our prototype:

1. Only symbols that introduce integer literal constants are considered as *candidate* enumerators. That is, we exclude symbols that introduce hexadecimal, octal, and character constants even though they are potential enumerators.
2. The conditional compilation directives of the preprocessor are ignored, implying that in our analysis there may be multiple declarations of the same symbol which may not occur during actual compilation.
3. A `#define` directive within a comment is considered to be valid, although it is ignored by compilers.

Table 2 presents some data to assess the implications of the above compromises. This table gives a distribution of macro symbols that introduce various types of constant literals in the include files of the SunOS 4.1.3_U1 library. 761 of the symbols are integral valued and are, therefore, potential enumerators. 533 (or 70%) of the integral valued symbols are candidate enumerators due to the exclusions of compromise one. Compromises two and three imply that our system processes more symbols than it should. Although we have no precise data, we have observed that the number of such spurious symbols is relatively small. Consequently, these spurious symbols have no significant affect upon our analysis.

5.3 Oracle

No oracles or expected data sets existed for the subject

Table 1. Summary of interesting statistics of the subject systems.

System	Number of .h files	Size in LOC of the .h files	Number of #defines	Number of candidate symbols
BTOOL	33	4937	428	44
EDGE	53	3842	368	217
FIELD	25	2175	52	12
GCC	57	13649	1090	160
GHOST-SCRIPT	128	11212	1305	299
SGEN	125	12230	1648	234
/usr/include	75	4918	1225	533
WPIS	66	7150	849	75

Table 2. Distribution of different usages of macro symbols in the files `/usr/include/*.h` of SunOS 4.1.3_U1.

Total number of include (.h) files	75
Total number of #defines	1225
Total number of #defines introducing constant literals	
Integer	533
Hexadecimal	169
String	59
Octal	49
Floating point	23
Character	10
Other types of #defines	382

systems. Hence, we manually constructed a set of expected data for each system using the following approach.

For each `.h` file, we extracted all of the integer valued symbols in the order in which they were declared, but with no intervening text or visual cues. We then partitioned the symbols into groups using the following rules: consecutively declared symbols were placed in the same partition if (a) they were lexically similar, such as `INPUT` and `OUTPUT`, or (b) they appeared related from their names based upon our programming and domain knowledge, such as `TRUE` and `FALSE`, and `YES` and `NO`.

5.4 Congruence measure

A heuristic is, by definition, "a speculative formulation serving as a guide in the investigation or solution of a problem[†]." It is very unlikely that any heuristic in general, and one for recovering enumeration types in particular, would lead to a precisely correct solution under all circumstances. Hence, the quality of a heuristic, i.e., its effectiveness in identifying the correct sets of enumeration types, may be measured in terms of the "degree of similarity" between the solution it generates and the corresponding ideal solution, such that a "closer" solution is assigned a higher value of effectiveness.

In our context, each system is already partitioned into files. An enumeration type recovery heuristic only partitions the set of symbols within each individual file. A measure of the effectiveness of a heuristic should summarize its performance over all files in a system. Thus, our measure of effectiveness consists of two parts:

1. A *measure of congruence* between two partitions of the same set of elements for quantitatively comparing the degree of similarity between the recovered and expected sets of enumeration types in a file.
2. A *weighted average of congruence measures* over all of the files in a system to summarize the performance of a heuristic over an entire system.

The congruence between two partitions A and B may be measured as a fraction of the total pairs of elements that are *similarly placed* in both of the partitions, where a pair of elements x and y are said to be similarly placed in A and B if the following conditions are satisfied: (a) if x and y appear in the same group in one partition then they appear in the same group in the other partition, and (b) if x and y appear in different groups in one partition then they appear in different groups in the other partition [17]. This may be defined more formally as follows. Let E be the set of all macro symbols in a program and let $\mathcal{T}$ be the set $E \times E$. A partition of E, say A, is essentially an equivalence relation over E such that $(x, y) \in A$ implies x and y are in the same group in A. Thus, $A \subseteq \mathcal{T}$. Similarly, $\bar{A} = \mathcal{T} - \mathcal{A}$ gives the pairs of elements not in the same group in partition A. The congruence measure $CM(A, B)$ is then defined as:

$$\mathrm{CM(A,B)} = \frac{|\mathrm{A} \cap \mathrm{B}| + |\bar{\mathrm{A}} \cap \bar{\mathrm{B}}|}{|\,\mathcal{T}\,|}$$

The congruence measure yields a value in the range of 0 to 1, where a higher value indicates a greater percentage of similarly placed elements between the partitions being compared.

The overall effectiveness, $Effectiveness(S, h)$, of a heuristic is defined as:

$$\frac{\sum_{f \in S} \mathrm{CM}(h(f), \mathrm{oracle}(f)) \times \mathrm{w}(f)}{\sum_{f \in S} \mathrm{w}(f)}$$

where S is a software system, h is a heuristic, f is a file in the system, $h(f)$ gives the partitions for file f generated by heuristic h, oracle(f) gives the corresponding partition for file f generated by the oracle, and w(f) gives the weight associated to file f.

We have chosen the weight function: $\mathrm{w}(f) = \mathrm{n}_f$, the number of *candidate* symbols in file f. This function increases the weight linearly with the size of the problem space. Alternatively, one may choose (a) a constant function $\mathrm{w}(f) = 1$ if the increase in weight is not desired, (b) a quadratic function $\mathrm{w}(f) = \mathrm{n}_f^2$ if a faster increase is desired, or (c) a logarithmic function $\mathrm{w}(f) = \log(\mathrm{n}_f)$ for a slow, yet positive, increase.

6 Discussion of experiment results

We now present an analysis of the data based upon the criteria of precision and impact. Since encapsulating a set of symbols into an enumeration type improves the readability and reliability of programs [22], these criteria enable us to evaluate the potential improvement in code quality that a heuristic offers.

The precision of various heuristics in recovering correct enumerations is assessed using the congruence measure introduced in Section 5.4. Since this measure is normalized, it does not give an indication of the size of the problem space. For example, a 90% congruence in recovering enumerations for a system does not indicate whether the system has 10 or 1,000 symbols.

The size of the problem space is useful for evaluating the expected impact of solving the problem. In our context, a 90% recovery on a space of 1,000 symbols would have a greater impact on improving code quality than a similar recovery on 10 symbols.

6.1 Discussion of precision

Table 3 summarizes the number of candidate symbols in each subject system and the $Effectiveness$ of each heuristic in recovering the correct sets of enumeration

[†] The American Heritage Dictionary, Second College Edition, 1985.

Table 3. Effectiveness of various heuristics in recovering enumeration types proposed by an oracle.

System	Candidate symbols	File	Visual cues: Blank Line	Visual cues: Line Comment	Visual cues: Block Comment	Visual cues: Any Visual Cue	Sequence Level I	Sequence Level II
BTOOL	44	0.52	0.86	0.75	0.38	0.76	0.75	0.63
EDGE	217	0.33	0.66	0.54	0.33	0.66	0.79	0.80
FIELD	12	0.72	0.89	0.89	0.72	0.89	0.31	0.42
GCC	160	0.63	0.94	0.83	0.79	0.89	0.89	0.83
GHOST-SCRIPT	299	0.72	0.86	0.92	0.76	0.93	0.40	0.40
SGEN	228	0.49	0.92	0.74	0.62	0.94	0.85	0.86
usrInc	533	0.35	0.74	0.73	0.52	0.98	0.78	0.85
WPIS	75	0.86	0.85	0.86	0.77	0.85	0.39	0.39
Minimum $Effectiveness$		0.33	0.66	0.54	0.33	0.66	0.31	0.39
Maximum $Effectiveness$		0.86	0.94	0.92	0.79	0.98	0.89	0.86

types as proposed by the oracle. We have not performed any significant statistical comparisons to determine which heuristic is "better" since we have only eight data points per heuristic. Instead, we use the following rules to qualitatively evaluate the heuristics.

A heuristic is considered good if:

Rule 1: It consistently performs better than the *File* heuristic.

Rule 2: Its $Effectiveness$ is consistently greater than some threshold.

Rule 3: Its $Effectiveness$ is consistently among the top two for each system.

Rule 4: Its range of $Effectiveness$, as defined by the minimum and maximum $Effectiveness$, is "better" than the others.

The data in Table 3 shows that:

- only the *Line Comment* and *Any Visual Cue* heuristics are considered good based upon Rule 1. We can also include the *Blank Line* heuristic if we are willing to accept a 1% error factor for the WPIS system.
- *Blank Line* and *Any Visual Cue* may be considered good using Rule 2 if a threshold of 66% is used.
- *Any Visual Cue* is the only heuristic that may be considered good based upon Rule 3. It is the only heuristic with an $Effectiveness$ among the top two for all of the systems except EDGE, for which it is among the top three.
- *Blank Line* and *Any Visual Cue* are also considered good based upon Rule 4, since they yield the two highest minimum and maximum $Effectiveness$.

These results suggest that partitioning the candidate symbols on the *Any Visual Cue* heuristic consistently gives as good or better results than any of the other heuristics. Further, simply partitioning on the *Blank Line* heuristic is considered good using three of the four rules.

WPIS is the only system for which none of the heuristics outperforms the *File* heuristic. Examination of the data in Table 1 reveals the cause: an average of 1.14 candidate symbols per file. Since we only group symbols declared in a file, an average of 1.14 symbols per file implies that the groupings due to the oracle cannot be very different from the *File* heuristic. This is reflected in Table 3 the 86% $Effectiveness$ of the *File* heuristic. Thus, for the WPIS system, the *File* heuristic is among the best heuristics.

6.2 Discussion of impact

In the absence of a single measure for impact, we use the distribution of size of groups created by various heuristics to make some qualitative comparisons. We present the distribution of only one subject system due to space constraints. This distribution gives an indication of the potential improvement in code quality that a heuristic gives, subject to the assumption that all of the groups it identifies are correct.

Table 4 gives the distribution of the size of the groups created by various heuristics for the `/usr/include/*.h` files of SunOS 4.1.3_U1. Each column, except the first, gives the distribution for a heuristic. The *File* heuristic provides a control in that it gives the number of files containing a particular number of candidate symbols. Since all other heuristics decompose the set of candidate symbols in a file, they all have (a) at least as many groups of size 1 as the *File* heuristic and (b) at least as many total number of groups as the *File* heuristic.

The total number of groups with at least two elements is an indicator of how many enumeration types may be created, since it is not meaningful to create an enumeration type with just one element. For the data in Table 4, the heuristic *Any Visual Cue* (labeled Any V/Cue) proposes the highest number of enumeration types, and hence may be considered better than others. The next best performer is the *Sequence Level II* heuristic (labeled Seq. II).

7 Conclusions

We have introduced the problem of identifying candidate enumeration types from C code. This is a sub-problem of the broader problem of object recovery. Prior research

Table 4. Distribution of the size of groups in partitions created by various heuristics for the files `/usr/include/*.h`.

Size of group	File	Comments		Blank	Any	Seq.	Seq.
		block	line	lines	V/Cue	I	II
1	9	16	24	45	53	129	135
2	4	6	10	10	13	12	12
3	2	6	9	10	17	14	15
4	4	4	4	8	10	5	7
5	1	1	2	2	1	5	4
6	2	2	7	3	6	3	3
7	1	3	2	4	4	3	5
8		1	1	5	6	2	4
9	2	3	2		2	4	2
10		1	1	3	3		
11	1	1	2		1	2	2
12	1		5	5	6	3	5
13	1	2	1				
14	1			1	2	1	2
15						1	1
16	1	1	1				
17		1		1	1	1	1
18	1	1	1		1	2	2
19		1	1	1	1	1	
23			1				
25		1					
29			1				
32	1	1	1				
33	1	1	1	1	1		
35	1		1				
36	1						
41	1						
43						1	
44	1	1	1				
137				1			
147	1	1					
Total number of groups							
	38	55	79	100	128	189	200
Number of groups of size > 1							
	29	39	55	55	75	60	65
Number of elements in groups of size > 1							
	524	517	509	488	480	404	398

on object recovery has assumed the existence of an encapsulated collection of data. Our work *extends* this line of research to the recovery of unencapsulated collections of data. This is an important step in reengineering legacy code into the object-oriented paradigm for programs written in languages that lack constructs for encapsulating related data. Our data indicates that about 70% of the symbolic constants introduced using `#define` directives have a value representable by the type `int`, and hence qualify to be part of an enumerated type (Table 2).

We have presented and experimentally evaluated a set of heuristics to partition the set of macro symbols into groups such that each group constitutes an enumerated type. The heuristics are: *Blank Line*, *Line Comment*, *Block Comment*, *Any Visual Cue*, *Sequence Level I*, and *Sequence Level II*. The first four heuristics are based upon the intuition that the visual cues programmers use to delineate blocks of related code may be used for partitioning the macro symbols as well. The first three heuristics partition the symbols based upon visual cues provided by blank lines, single-line comments, and multiple-line comments, respectively, whereas the fourth partitions the symbols based upon any of the first three heuristics. The *Sequence Level I* heuristic groups successive symbols that introduce an incrementing or decrementing sequence of integer values. It is based upon the intuition that enumerated constant symbols are frequently assigned successive numeric values. The last heuristic first creates partitions using the *Any Visual Cue* heuristic and then decomposes each group again using the *Sequence Level I* heuristic.

We performed a quantitative analysis of the effectiveness of these heuristics on a set of eight moderate-sized, real-world programs. The data suggests a 66% to 98% match between the enumerated types proposed by the *Blank Line* and *Any Visual Cue* heuristics and those proposed by a programmer. In contrast, the *Sequence Level I* heuristic—the first heuristic we thought of—produces only a 31% to 89% match, which is not a very good performance in comparison to the 33% to 86% match achieved by simply creating a single group from the macro symbols in a file.

We excluded symbols introducing hexadecimal, octal, and character literal constants based upon our initial bias towards the *Sequence Level I* heuristic. We had anticipated the need to develop a different set of heuristics for these literal constants. Extrapolating from our preliminary results, however, it appears that the *Blank Line* and *Any Visual Cue* heuristics will perform equally well for these constants and that a separate heuristic may, in fact, not be needed.

Acknowledgments

The authors thank A. J. Ciallella, K. Covington, and D. Hillard for their proofreading and helpful suggestions in the development of this paper. We also thank the anonymous referees for their comments that helped substantially improve this paper.

References

[1] G. Canfora, A. Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data types. In *Proceedings of the Working Conference on Reverse Engineering*, pages 73–82. IEEE CS Press, May 1993.

[2] G. Canfora, A. Cimitile, M. Tortorella, and M. Munro. A precise method for identifying reusable abstract data types in code. In *International Conference on Software Maintenance*, pages 404–413. IEEE CS Press, Sept. 1994.

[3] S. C. Choi and W. Scacchi. Extracting and restructuring the design of large systems. *IEEE Software*, 7(1):66–71, Jan. 1990.

[4] A. Cimitile, M. Tortorella, and M. Munro. Program comprehension through the identification of abstract data types. In *Proceedings of the 3rd Workshop on Program Compre-*

hension, pages 12–19. IEEE CS Press, Nov. 1994.

[5] N. Dale, C. Weems, and M. Headington. *Programming and Problem Solving with C++*. D. C. Heath, 1996.

[6] S. Horwitz, J. Prins, and T. Reps. Integrating non-interfering versions of programs. In *Proceedings of the Fifteenth Annual ACM Symposium on Principles of Programming Languages, San Diego*, pages 133–145, 1988.

[7] D. H. Hutchens and V. R. Basili. System structure analysis: Clustering with data bindings. *IEEE Trans. Softw. Eng.*, pages 749–757, Aug. 1985.

[8] B. W. Kernighan and D. M. Ritchie. *The C Programming Language*. Prentice Hall, Second edition, 1988.

[9] A. Lakhotia. A unified framework for expressing architecture recovery techniques. *Journal of Systems and Software*, 1996. (to appear).

[10] S. Liu and N. Wilde. Identifying objects in a conventional procedural language: An example of data design recovery. In *Proc. IEEE Conference on Software Maintenance*, pages 266–271, Nov. 1990.

[11] P. E. Livadas and T. Johnson. A new approach to finding objects. Technical Report SERC-TR-63-F, Software Engineering Research Center, Computer and Information Sciences Department, University of Florida, June 1993.

[12] H. A. Müller and J. S. Uhl. Composing subsystem structures using (K,2)–partite graphs. *Proceedings of the Conference on Software Maintenance*, pages 12–19, Nov. 1990.

[13] R. M. Ogando, S. S. Yau, S. S. Liu, and N. Wilde. An object finder for program structure understanding in software maintenance. *Journal of Software Maintenance Research and Practice*, 6(5), Sep-Oct 1994.

[14] C. Ong and W. T. Tsai. Class and object extraction from imperative code. *J. Object Oriented Programming*, pages 58–68, Mar–Apr 1993.

[15] F. N. Paulisch and W. F. Tichy. EDGE: An extendible graph editor. *Software–Practice and Experience*, 20(S1):63–88, June 1990.

[16] T. W. Pratt and M. V. Zelkovitz. *Programming Languages: Design and Implementation*. Prentice Hall, Third edition, 1996.

[17] W. M. Rand. Objective criteria for the evaluation of clustering methods. *Journal of the American Statistical Association*, 66(336):846–850, Dec. 1971.

[18] S. P. Reiss. Connecting tools using message passing in the Field environment. *IEEE Software*, 7(4):57–66, July 1990.

[19] T. Reps and T. Teitelbaum. *The Synthesizer Generator: A System for Contructing Language-Based Editors*. Springer-Verlag, New York, NY, 1988.

[20] T. Reps and T. Teitelbaum. *The Synthesizer Generator Reference Manual*. Springer-Verlag, New York, NY, third edition, 1988.

[21] S. Rugaber, K. Stirewalt, and L. M. Wills. The interleaving problem in program understanding. In *Second Working Conference on Reverse Engineering*, pages 166–175. IEEE Computer Society Press, July 1995.

[22] R. W. Sebesta. *Concepts of Programming Languages*. Benjamin/Cummings, 1989.

[23] I. Silva-Lepe. An empirical method for identifying objects and their responsibilities in a procedural program. In *Technology of Object-Oriented Languages and Systems Europe Conference*, pages 136–149, Versailles, France, 1993. Prentice-Hall. Also available as technical report NU-CCS-93-2, Northeastern University.

[24] B. Stroustrup. *The C++ Programming Language*. Addison-Wesley, Second edition, 1991.

[25] A. S. Yeh, D. R. Harris, and H. B. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Proceedings of the 2nd Working Conference on Reverse Engineering*, pages 227–236. IEEE CS Press, July 1995.