Retrieving Reusable Software by Sampling Behavior

ANDY PODGURSKI
Case Western Reserve University
and
LYNN PIERCE
Allen-Bradley Company

A new method, called *behavior sampling*, is proposed for automated retrieval of reusable components from software libraries. Behavior sampling exploits the property of software that distinguishes it from other forms of text: executability. Basic behavior sampling identifies relevant routines by executing candidates on a searcher-supplied sample of operational inputs and by comparing their output to output provided by the searcher. The probabilistic basis for behavior sampling is described, and experimental results are reported that suggest that basic behavior sampling exhibits high precision when used with small samples. Extensions to basic behavior sampling are proposed to improve its recall and to make it applicable to the retrieval of abstract data types and object classes.

Categories and Subject Descriptors: D.2.2 [**Software Engineering**]: Tools and Techniques; D.2.7 [**Software Engineering**]: Distribution and Maintenance; D.2.m. [**Software Engineering**]: Miscellaneous—*reusable software*; H.3.1 [**Information Storage and Retrieval**] Content Analysis and Indexing; H.3.3 [**Information Storage and Retrieval**]: Information Search and Retrieval

General Terms: Algorithms, Design, Experimentation, Human Factors

Additional Key Words and Phrases Behavior sampling, software libraries, software retrieval, software reuse

1. INTRODUCTION

The high cost of software development dictates that software components and related artifacts like requirements, designs, and test plans be *reused* when-

This paper is a revised and extended version of [31], which was presented at the *Proceedings of the 14th International Conference on Software Engineering*, May 11–15, 1992, sponsored by the Institution of Engineers, Australia; the IEEE Computer Society; the Association for Computing Machinery/SIGSOFT; the Institution of Radio and Electronics Engineers, Australia; and the Australian Computer Society. A. Podgurski's research was supported by NSF Research Initiation Award CCR-9009375.

Authors' addresses. A. Podgurski, Computer Engineering and Science Department, Case Western Reserve University, Cleveland, OH 44143; L. Pierce, Allen-Bradley Company, 747 Alpha Drive, Highland Heights, OH 44143.

ever possible. Reuse can expedite software development, reduce its cost, and improve software reliability [5, 6, 8, 14, 19, 21, 22, 24, 28, 36, 38]. However, there are several problems that must be resolved in order to achieve significant reuse. One of these is the problem of *component retrieval*. To facilitate reuse, it is sensible to maintain a library of reusable components. Convenient use of a large software library requires an efficient method for retrieving components satisfying given requirements [1, 4, 5, 7, 8, 10–13, 16, 19, 21, 26, 27, 33]. A number of automated methods for retrieving ordinary text have been adapted for software retrieval. None of these is completely satisfactory for retrieving *executable* components, however. The essential reason is that the methods' criteria for identifying relevant components are related only weakly to execution behavior.

Ordinary text-retrieval methods do not exploit the property that fundamentally distinguishes software from other kinds of text: Software can be *executed* to *transform* inputs to outputs. We propose a new method for automated retrieval of reusable components, called *behavior sampling*, that does exploit this property. In the most basic form of behavior sampling, a person (hereafter called the *searcher*) who seeks a routine satisfying given requirements first specifies the routine's interface and randomly chooses a small sample of inputs[1] from the required routine's *operational input distribution op* or from an approximation to *op*. (The probability distribution *op* describes the relative frequency with which different inputs to the required routine will arise in its operating environment(s)). The searcher then computes the required output corresponding to each of these inputs, manually if necessary. All library routines whose interface is compatible with that specified by the searcher are automatically identified and executed on the input sample. Any routine whose output over the sample matches the output specified by the searcher is retrieved, along with its documentation, for the searcher to inspect. Of course, behavior sampling is not directly applicable to retrieving nonexecutable artifacts such as requirements, designs, and test plans.

It can be shown (see Section 4) that a routine that satisfies the searcher's requirements on a nonnegligible random sample of inputs from the operational distribution *op* is likely to satisfy them on any input chosen randomly from *op*. The premise of behavior sampling is that *a routine that is likely to satisfy the user's requirements on an input chosen randomly from op is likely to be relevant to those requirements*. This hypothesis cannot be proven mathematically, because its truth depends on programming practice. We conducted an experiment to test the hypothesis.[2] In the experiment, basic behavior sampling was used 40 times to find routines in a composite library of 252 utility routines. The experiment supported our hypothesis: Behavior sampling retrieved *only* relevant routines when input samples of size 12 were used. Moreover, the results suggest that behavior sampling performs well with even smaller input samples.

[1]By *input* we mean an assigment of values to all of a program's input variables (note that we view a file as a value); we call the value assigned to any variable an *element* of the input.
[2]The experiment reported here expands upon that reported in [31].

Basic behavior sampling has some significant but rectifiable limitations. First, it cannot retrieve components for which it is impractical to determine appropriate output. Second, it cannot retrieve components whose behavior is "close" but not identical to that required by the searcher. Third, it does not facilitate the (direct) retrieval of *abstract data types* or their close relatives *object classes*, that is, data types defined by the set of services (operations) they provide and by the interrelationships of those services. We address these limitations by proposing two extensions to basic behavior sampling. One of these permits searchers to determine the criteria used for retrieving a component, by providing a programmed acceptance test. The other extension supports the retrieval of abstract data types and object classes.

In Section 2 we discuss existing retrieval methods and their limitations. The implementation of basic behavior sampling is outlined in Section 3. In Section 4 the probabilistic basis for behavior sampling is described. Our experimental evaluation of basic behavior sampling is described in Section 5. In Section 6 our extensions to basic behavior sampling are presented. Finally, concluding remarks are made in Section 7.

2. EXISTING RETRIEVAL METHODS AND THEIR LIMITATIONS

Automated retrieval of software components is usually achieved with *text-retrieval* methods [4, 7, 11, 13, 27, 32, 33, 35]. Such methods search for lexical or syntactic objects associated with a component, such as words or phrases. We shall call these objects *terms*. Since scanning a large amount of text is slow, many text-retrieval systems index a collection of texts using terms found in the texts or assigned from a controlled vocabulary. Index terms are selected on such bases as subjective judgment, term frequencies in text, and how often searchers associate particular terms and objects. Text-retrieval systems use aliases, synonyms, stems, broader terms, and other related terms to enhance the *recall* of a query, that is, the percentage of all relevant texts retrieved. They use phrases, lexical affinities, term weights, and narrower terms to increase *precision*, the percentage of retrieved texts that are relevant. Some methods allow a searcher to use Boolean operators to form a complex query.

An important class of controlled term vocabularies are those associated with *classification schemes*, which are used to structure libraries [3, 4, 7, 11, 12, 16, 29, 32–34]. Most classification schemes for software libraries consist of a hierarchy of inclusion relationships between categories of components. Prieto-Díaz and Freeman [33] described an alternative scheme, called *faceted classification*, with which software components are classified by choosing values (terms) for a set of attributes called *facets*. Faceted classification is more flexible than hierarchical schemes. Some classification schemes for software components are based on implementation dependences between components, such as inheritance and reference [3, 11, 12, 16, 29, 37].

While text-retrieval methods are valuable tools for finding reusable components, they have some serious limitations. Studies suggest that there is an inherent trade-off between the precision and recall of text-retrieval methods

[35]. Because people use a great variety of terms to refer to the same thing, it is often necessary to make extensive use of aliases, synonyms, etc., to achieve acceptable recall [15], which, in turn, reduces precision. This problem is exacerbated with software components, whose function is often obscure and not amenable to precise description by a few terms. Indeed, one of us (Pierce) observed programmers in an industrial setting quickly become discouraged with the imprecision of a faceted classification scheme for reusable components and cease to use it. Prieto-Díaz [32] found that this problem was ameliorated by devising domain-specific classification schemes, but this is obviously precluded for general-purpose libraries.

Some retrieval systems use artificial-intelligence techniques to represent and reason about knowledge of component semantics, applications domains, user ability, and natural language [3, 10, 20, 37, 41]. Such systems may exhibit more precision and recall in a particular setting than text-retrieval systems do. However, this advantage comes at the cost of generality, since considerable manual domain analysis is currently required to build powerful retrieval systems using AI techniques. Fully automated understanding of requirements and of reusable components would seem to entail the solution of outstanding problems of knowledge representation, natural language understanding, and program analysis, and thus does not seem imminent.

Aiken [2] proposed a knowledge-based approach for selecting and testing reusable code that is somewhat similar to behavior sampling. In this approach, an expert system is used to select applicable modules and to synthesize candidate designs. Candidate design models are then evaluated using discrete event simulation.

In principle, it is possible for an automated retrieval system to input a formal requirements specification and to retrieve relevant software from a library of formally specified components by invoking a theorem prover to determine if component specifications satisfy the requirements. Were this practical, it would be an elegant solution to the retrieval problem. Unfortunately, the current state of theorem-proving technology [42] does not permit it. A prototype of a system that uses a theorem prover to reduce interaction with a searcher is described in [23].

3. THE MECHANICS OF BASIC BEHAVIOR SAMPLING

An implementation of behavior sampling must provide a mechanism by which the searcher can specify acceptable interfaces for routines satisfying the searcher's requirements. This mechanism must permit the searcher to specify the number, types, and modes (input/output) of parameters of suitable routines; parameters need not be named. We shall refer to the searcher's specification of these attributes as the *target-interface specification*. We call library routines whose interface is compatible with the target-interface specification *candidate routines*. An implementation of basic behavior sampling determines if a routine is a candidate by comparing an interface specification for the routine to the target-interface specification. An implementation might also attempt to instantiate *generic* routines automatically in order to create candidates. The behavior of candidate routines is evaluated by executing them on the input sample. Since the order of a routine's parameters is not usually an important criterion for reusability, a routine R is a candidate if there is a one-to-one correspondence between the parameters of the target-interface specification and the formal parameters of R and if this correspondence respects types and modes. For each such correspondence σ and each sample input I, R is executed with actual parameters obtained by using σ to map the elements of I to the formal parameters of R. In basic behavior sampling, a candidate routine's output is tested for equality with searcher-provided output to determine if the routine should be identified to the searcher.[3]

Preprocessing the interface specifications of library routines permits the construction of a data structure that facilitates efficient identification of candidate routines. To maximize the efficiency of behavior sampling, library routines must be compiled and linked to the behavior-sampling system, so that they can be invoked directly. In basic behavior sampling, a routine that fails to produce the required output on some sample input need not be executed on the remainder of the sample. An implementation of behavior sampling must, of course, handle exceptions arising during execution of library routines. In addition, timed interrupts can be used to handle nonterminating routines. Presumably, such occurrences should be treated as inappropriate behavior.

It seems useful to allow the searcher to specify alternative target interfaces for a single search, since an implementation of behavior sampling may search more efficiently if provided with alternatives initially than if invoked separately for each alternative. It also seems useful to allow the searcher to restrict search to a specified subset of a large library, for example, one selected using text-retrieval methods.

4. THE PROBABILISTIC BASIS FOR BEHAVIOR SAMPLING

An operational input distribution for a routine R is defined by the sequence $r_1, r_2, \ldots, r_N$ of runs (executions) of R that occurs in an execution environment E during some finite period of interest $[t_1, t_2]$. Let $\mathscr{D}$ be the input domain for R, and let $I(r_i)$ denote the input to R in run r_i, $1 \le i \le N$. The operational input distribution for R with respect to E and $[t_1, t_2]$ is the probability function op: $\mathscr{D} \to [0,1]$, defined as follows:

$$op(d) = \frac{|\{i: 1 \le i \le N \wedge d = I(r_i)\}|}{N}.$$

op characterizes the relative frequency with which different inputs arise in E between time t_1 and time t_2.

Suppose that a searcher requires a routine that computes a total function f: $\mathscr{D} \to X$. Let R be any candidate routine; that is, suppose R computes a

[3]Two floating-point numbers are treated as "equal" if their absolute difference is less than a small "fuzz" value.

function $g: \mathscr{D} \rightarrow X$. Let *op* be the operational distribution on $\mathscr{D}$ that characterizes the desired routine's future usage. Let p be the probability that f and g agree on an input chosen randomly from *op*:

$$\begin{aligned} p &= Pr(\{d \in \mathscr{D}: f(d) = g(d)\}) \\ &= \sum_{d:\, f(d)=g(d)} op(d). \end{aligned}$$

The probability that f and g agree on a random sample of n inputs drawn independently from *op* is p^n, an exponentially decreasing function of n. Even for values of p as high as 0.9, $p^n < 0.1$ for $n \geq 22$. Thus, unless the probability is high that a given library routine satisfies a searcher's requirements on an input drawn randomly from *op*, the routine is unlikely to satisfy those requirements on a sample of nonnegligible size drawn randomly from *op*. We contend that if the former probability is even moderately high, this constitutes sufficient justification to retrieve the library routine and to inspect it to determine if it meets the searcher's requirements or can be modified to do so.

The input distribution from which sample inputs are drawn for behavior sampling must usually be an approximation to a required routine's future operational distribution, since the latter distribution is seldom known with certainty. *Fortunately, very rough approximations to the operational distribution can be used effectively for behavior sampling.* Even choosing inputs *subjectively* can be successful. All distributions are not equally appropriate though. Consider, for example, using behavior sampling to find a routine that implements an equality predicate on strings over the English alphabet. Presumably, most operational input distributions for such routines do not assign minuscule probability to the set of pairs of identical strings. However, if one were to select randomly and *uniformly* an input sample of size 25 from the set of all pairs of strings of length ≤ 10, the probability that the sample would contain one or more pairs of identical strings is less than 1.7×10^{-13}. Thus, if behavior sampling were used with such a sample and if there were a library routine that computed the binary predicate on strings whose value is always *false*, this routine would almost certainly be retrieved. On the other hand, any input sample that contained at least one pair of identical strings would exclude this routine.

5. EXPERIMENTAL RESULTS

It is useful to evaluate methods for retrieving software components in terms of three measures often used to evaluate text-retrieval methods: precision, recall, and efficiency. Precision and recall have been defined in Section 2. Efficiency refers here to the time and space required by a retrieval method. In [31] we reported the results of an experiment conducted to assess the precision and efficiency of basic behavior sampling. In that experiment, basic behavior sampling was used with small input samples to find 20 randomly selected routines in a library of 145 utility routines. The method exhibited high precision. Moreover, the average number of candidate routines per retrieval was only 12.75, which suggests that the method can be implemented efficiently. To evaluate the precision and efficiency of basic behavior sampling further, we extended our earlier experiment by including additional libraries. In the remainder of this section, we describe the extended experiment and its results.

5.1 Purpose

The precision of a software-retrieval mechanism determines how much effort the searcher must expend to determine which retrieved components, if any, are usable. The precision of behavior sampling naturally tends to increase with the size of the input sample. However, the method's cost also increases with sample size, and its efficiency decreases correspondingly. Thus, the utility of the method depends on the sample size required to achieve adequate precision. The primary purpose of our experiment was to assess the precision of basic behavior sampling when it is used with small input samples. A secondary purpose was to evaluate its efficiency.

5.2 Method

An application of behavior sampling is characterized by the searcher, the searcher's requirements, the operational input distribution, and the library. Ideally, behavior sampling would be assessed by evaluating its performance in a number of randomly chosen applications. However, such an experiment would be difficult logistically, so we conducted a simpler one instead. A sample of requirements, including target-interface specifications, was obtained by selecting 40 routines from the three libraries shown in Table I and then inferring the routines' requirements from their documentation.[4] Sample inputs were generated pseudorandomly. Corresponding required outputs were computed by executing the selected routines. (Thus, our experiment did not address the difficulty of determining required outputs manually.) Finally, a composite library with 252 routines was formed from the three libraries in Table I, and the composite library was searched for routines satisfying the sample requirements.

Of the 40 routines from which requirements were inferred, 20 were selected from the VAX-11 C Library, 10 from the VMS Mathematics Library, and 10 from the VMS String-Manipulation Library. These routines were selected randomly from among those routines whose arguments and results are all denoted by explicit (formal) parameters. The parameters of the latter routines are integers, floating-point numbers, characters, strings, and variants of these. Routines that reference or modify the state of the file system, operating system, or C run-time system were excluded, to simplify the task of determining if a candidate routine produces the required output. (Note that this determination is possible, but more difficult, for the excluded routines.) The 40 selected routines are listed in Table II.

[4]The VMS Mathematics Library actually contains 136 routines, but 61 of these were excluded from our experiment. Routines using the **G_floating** and **H_floating** types were excluded if functionally equivalent routines using the **F_floating** or **D_floating** types existed.

Table I. Libraries Used in the Extended Experiment

Library name	*Number of routines*
VAX-11 C Library	145
VMS Mathematics Library	75
VMS String-Manipulation Library	32

Table II. Routines for which Requirements were Inferred

	VAX-11 C routines	
acos	atan	atof
atol	cos	ecvt
fabs	gcvt	isdigit
isupper	log	modf
sin	sqrt	strchr
strcmp	strlen	strncpy
strspn	_tolower	
	VMS mathematics routines	
mth$acos	mth$asin	mth$atan
mth$atand2	mth$cdabs	mth$clog
mth$datand	mth$dcos	mth$sinh
mth$tand		
	VMS string routines	
str$add	str$compare_eql	str$element
str$find_first_in_set	str$find_first_substring	str$len_extr
str$mul	str$position	str$pos_extr
str$translate		

For each library search, a sample of 12 inputs was selected randomly. Numbers and characters were generated pseudorandomly. Strings were chosen randomly from the Geauga County, Ohio, telephone book. When the search requirements called for a routine solving a binary decision problem, half of the selected inputs were positive instances of the problem, and half were negative.[5] Similarly, when the required behavior was conditioned on whether a certain relationship holds between two input strings (e.g., **str$find_first_substring**), half of the input pairs realized the relationship and half did not. In any case, the required outputs corresponding to the sample inputs for a routine were computed using the routine itself.

A C-language program was written to implement the experiment.[6] For each set of requirements inferred from a routine in Table II, this program performed the following tasks: It identified candidate routines by comparing the target-interface specification to interface specifications for the routines in the composite library; it executed candidate routines on each input from the

[5]This was a larger percentage of positive instances than was selected in the experiment reported in [31].

[6]This program was not written for general-purpose use.

sample selected for the search requirements; and for every sample, it recorded the number of times each candidate routine produced the required output.

5.3 Results

The most important data from our experiment are summarized in Table III, where the ith numbered column of the row r_R for routine R contains the number of routines from the composite library that are not functionally equivalent to R, but whose output matched R's on exactly i inputs from the sample S selected for R's requirements (empty squares denote zero). If j is the number of the leftmost column of row r_R such that all entries in columns $j, j + 1, \ldots, 12$ are zero, then no library routine that is not functionally equivalent to R agrees with R on as many as j elements of S. Thus, any subset of S containing j or more elements will distinguish any library routine that satisfies R's requirements from one that does not. We therefore call j the *distinguishing sample size* for the search involving R's requirements. This quantity characterizes the precision exhibited by behavior sampling in that search.

A histogram of the distinguishing sample sizes for the experiment is shown in Figure 1. In this figure the bar to the right of a routine name indicates the distinguishing sample size for the search involving the routine's requirements. Note that in no case did the distinguishing sample size exceed nine. In 26 of the 40 searches (65 percent), the distinguishing sample size was one. In 33 of the 40 searches (83 percent), the distinguishing sample size was at most five. The averaged value of the distinguishing sample size was only 2.6.

Figure 2 is a histogram displaying, for each routine R in Table II, the number of candidate routines found during the search for routines satisfying R's requirements. This number ranged from 1 to 44, the average being 17.95.

5.4 Analysis

The results of our experiment suggest that basic behavior sampling is very precise when used with small input samples. The method was perfectly precise in the experiment, in which input samples of size 12 were used. The distribution of distinguishing sample sizes suggests that the method may often be precise with significantly smaller samples. Since the efficiency of basic behavior sampling is inversely related to the sample size, these results also provide evidence that the method is efficient. The average number of candidate routines per search provides further evidence for the efficiency of the method; since this number was only 17.95, only a small percentage of the routines in the composite library needed to be executed on most searches. If this percentage is representative of other libraries, behavior sampling may prove widely applicable.

The results of our experiment, though encouraging, do not demonstrate that behavior sampling is generally precise or efficient. Our experiment did not account for possible user difficulty in determining required outputs corresponding to sample inputs, nor did it address the question of whether behavior sampling is sensitive to differences between operational input distri-

Table III. Summary of experimental data—the *i*th column of the row associated with routine *R* contains the number of routines from the composite library that are not functionally equivalent to *R* but whose output matched *R*'s on exactly *i* elements of the input sample used to find routines satisfying *R*'s requirements; empty squares denote zero

	1	2	3	4	5	6	7	8	9	10	11	12
acos												
atan												
atof												
atol												
cos												
ecvt		1										
fabs		1										
gcvt												
isdigit					1	7	1					
isupper					1	11						
log												
modf			1									
sin												
sqrt												
strchr	1					1						
strcmp	2	1	2									
strlen												
strncpy												
strspn		5	1			1						
_tolower						1						
mth$acos												
mth$asin												
mth$atan												
mth$atand2	2											
mth$cdabs												
mth$clog												
mth$datand												
mth$dcos												
mth$sinh												
mth$tand				1								
str$add												
str$compare_eql												
str$element												
str$find_first_in_set	4						1					
str$find_first_substring												
str$len_extr												
str$mul												
str$position	1											
str$pos_extr	1	3		1				1				
str$translate												

butions. Finally, it remains to be seen whether behavior sampling works as well with other types of software libraries. Further experimentation is clearly advisable.

6. EXTENSIONS TO THE BASIC APPROACH

6.1 Flexible Assessment of Routine Output

To use basic behavior sampling, the searcher must determine required output corresponding to sample inputs. As in software testing, this is often practical. However, experience with testing indicates that for some requirements this determination is quite difficult [39]. For example, determining if a compiler's

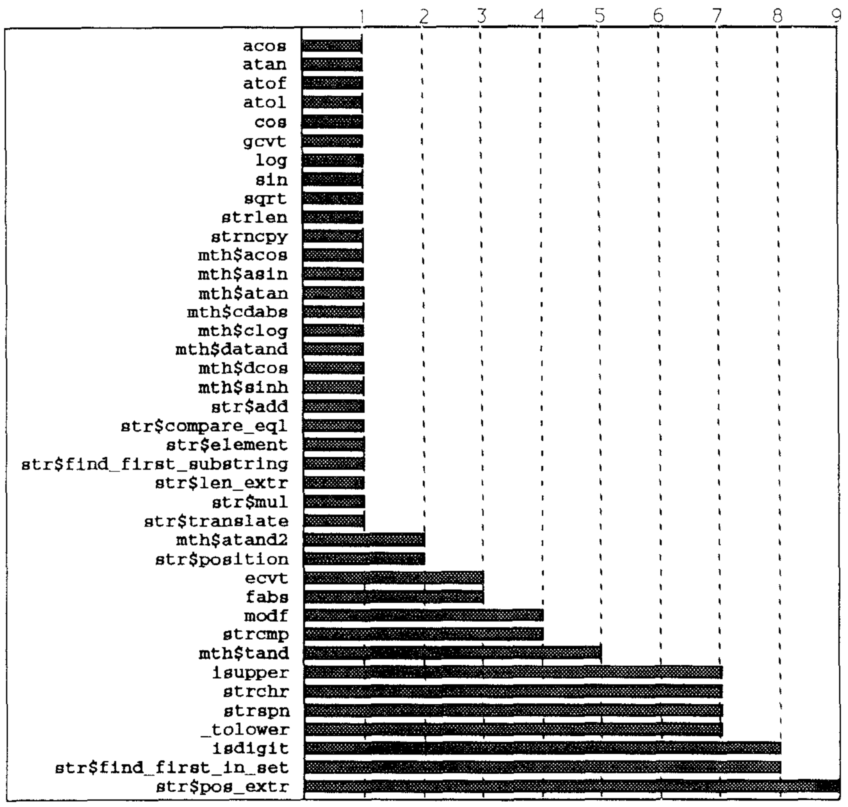

Fig. 1. Histogram of distinguishing sample sizes: The bar to the right of a routine name indicates the distinguishing sample size for the search involving the routine's requirements.

output is correct can be nearly impossible. One of the solutions adopted in testing is to check whether output satisfies certain conditions that are necessary or sufficient, but not both, for correctness. For example, a compiler's output (hopefully not another compiler!) might be checked by executing it on some test cases. There is no reason why basic behavior sampling cannot be extended to permit checks of this sort. The most flexible extension is to allow the user to provide a program that is executed during the search to determine if the output of a candidate routine is acceptable. Searchers can then program any acceptance test they want.

Allowing searchers to program an acceptance test for output also permits routines to be retrieved that do not produce the required output but can be modified easily to do so. For example, a numeric routine whose output always differs from the required output by a fixed constant could be used after minor modification, but would not be found with basic behavior sampling. A searcher could easily program a test for whether the output corresponding to an input sample differed from the required output by some constant. It is not possible for the implementer of a behavior-sampling retrieval system to anticipate

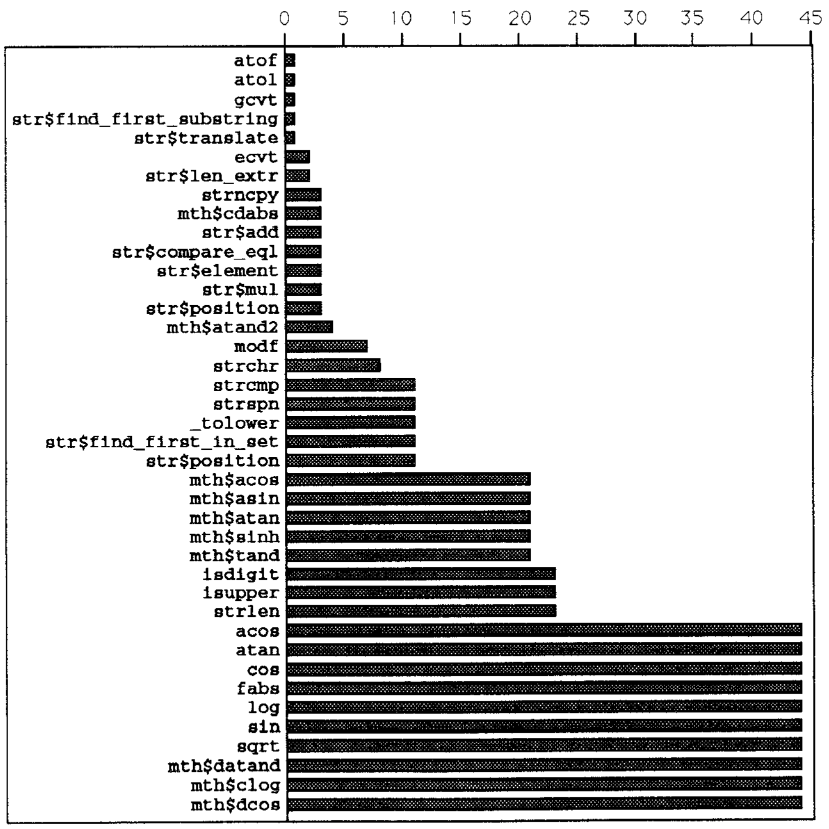

Fig. 2. Histogram showing the number of candidate routines found on each search in the experiment. Searches are denoted by the name of the routine from which requirements were inferred.

what the searcher will view as "close enough." For example, the difference between two vectors of real numbers can be measured using Euclidean distance or the maximum of the component differences. Which metric is appropriate for evaluating a routine's output depends on the intended use of the routine and can be decided only by the searcher.

6.2 Interface Compatibility and Abstract Data Types

Often, a searcher's requirements for a component are abstract and do not uniquely identify an appropriate physical interface for the component. There may be several reasonable but mutually incompatible ways to specify a component's interface. For example, a parameter representing a set may be implemented as a bit vector, linked list, tree, or hash table. Thus, a software library may contain components that satisfy a searcher's abstract requirements, but whose interfaces are incompatible with the searcher's target-interface specification. Such components cannot be found with basic behavior sampling. This problem is related to those that motivate the use of *informa-*

tion hiding [30] and *abstract data types* (*ADTs*) [17, 18, 25] in software design. Hence, it might be redressed by structuring a software library as a collection of ADTs, with the interface to each ADT specified in terms of other ADTs instead of physical data representations. The *class libraries* whose development is encouraged by object-oriented programming have essentially this form. For these reasons, we now consider the extension of behavior sampling to the retrieval of ADT implementation and object classes. We call this extension *ADT behavior sampling*.

ADT behavior sampling searches a library of ADT implementations for an implementation of a *target ADT*. The searcher specifies the target ADT by describing the interfaces (signatures) of the operations it provides and by identifying implementations for any other ADTs referenced in the operation interfaces. Naturally, such implementations may themselves be found using ADT behavior sampling. As in basic behavior sampling, the searcher provides a sample of appropriate input/output behavior that must be exhibited by implementations before they are retrieved. In contrast to basic behavior sampling, however, these behaviors are described in terms of *interactions* of an ADT's operations, because the operations of an ADT are often defined in terms of one another and cannot be evaluated separately. For example, the semantics of the stack operation *Pop* is usually defined in terms of the stack operations *Push* and *Top*; to find an implementation of *Pop* using behavior sampling, implementations of *Push* and *Top* are required.

We briefly review some definitions concerning abstract data types [40]. An abstract data type is denoted by a *signature* $\Sigma = (S, F)$, where $S = sorts(\Sigma)$ is a set of sorts (type identifiers) and $F = opns(\Sigma)$ is a set of function symbols. Each function symbol $f \in F$ has an associated type $type(f)$, which is a finite, nonempty sequence of elements of S. A *signature isomorphism* σ from Σ onto a signature $\Sigma' = (S', F')$ is a pair consisting of a bijection σ_{sorts} from S onto S' and a bijection σ_{opns} from F onto F' such that for all $f \in F$, if $type(f) = (s_1, s_2, \dots, s_n, s_{n+1})$, then $type(\sigma_{opns}(f)) = (\sigma_{sorts}(s_1), \sigma_{sorts}(s_2), \dots, \sigma_{sorts}(s_n), \sigma_{sorts}(s_{n+1}))$. Informally, two signatures are isomorphic if they are equivalent up to naming. A Σ-*algebra* A is a pair consisting of a family $\{A_s\}_{s \in S}$ of nonempty sets indexed by S and a family $\{f^A\}_{f \in F}$ of functions indexed by F such that if $type(f) = (s_1, s_2, \dots, s_n, s_{n+1})$ then f^A is a function from $A_{s1} \times A_{s2} \times \cdots \times A_{s_n}$ into $A_{s_{n+1}}$. It is easily seen that if A is a Σ-algebra and Σ is isomorphic to Σ' then A is also a Σ'-algebra. Informally, a Σ-algebra A is *term generated* if for each $s \in S$ and each $a \in A_s$ there is a variable-free term t over Σ that denotes a. For our purposes, an *abstract data type* of signature Σ is the class of all term-generated Σ-algebras that satisfy certain semantic properties characterizing the type. An *algebraic specification* consists of a signature Σ and a set E of axioms (first-order equational formulas) over Σ. The axioms of an algebraic specification express the semantic properties of an abstract data type.

To employ ADT behavior sampling, the searcher must provide items analogous to those provided in basic behavior sampling. The searcher must specify a signature Σ for the target-ADT T. A *candidate* implementation for T is an ADT implementation whose signature is isomorphic to Σ. The searcher also randomly chooses a sample $\mathscr{S}$ of operational *access computations* on T, for use in assessing the behavior of candidate implementations. An access computation produces a value of a type for which the searcher already possesses an implementation. Finally, the searcher provides a program schema P describing an algorithm that determines if a candidate implementation behaves acceptably when it is used to carry out the computations in $\mathscr{S}$. When constructing P, the searcher describes access computations on T using symbolic terms obtained by composing the operation and constant symbols of $opns(\Sigma)$, because the searcher does not know whether or how T is implemented in the library.[7] The library is automatically searched for ADT implementations I having a signature or subsignature Σ' that is isomorphic to Σ.[8] Each signature isomorphism $\sigma:\Sigma \rightarrow \Sigma'$ is used to map terms over Σ that occur in P to terms over Σ', which can be evaluated using the implementation I corresponding to Σ'. The program $\sigma^*(P)$ obtained by instantiating P in this way is automatically invoked to determine whether I is acceptable. If so, I and σ are identified to the searcher for inspection.

A sample of operational access computations can be obtained by analyzing how the program requiring an implementation of the target ADT would invoke it when executed on operational inputs. The reason only access computations are used to evaluate the behavior of candidate implementations is that determining whether a computed value is correct requires accepted implementations for operations (such as an equality test) on that value's type.

Example: *Ordered Sets*. The abstract data type *Order_Set_Type* represents finite sets of elements ordered by the value of a key. One signature for this ADT comprises the four sorts *Ordered_Set*, *Element*, *Key*, and *Boolean* and the following nine operations:[9]

(1) *Newset*: This is a nullary function (constant) that returns the empty set.
(2) *Insert*(S, x): This operation returns a set having as elements x and all the elements of S.
(3) *Delete*(S, x): This operation returns a set having all of the elements of S except x.
(4) *Empty*(S): This operation returns *true* if S contains no elements, and returns *false* otherwise.
(5) *Search*(S, k): This operation returns an element of set S having key k if such an element exists, and returns *NIL* otherwise.
(6) *Minimum*(S): This operation returns the element of a nonempty set S with the least key value.

[7]It is because of this use of terms over Σ that we call P a "program schema" and not a "program."
[8]The searcher may require that a signature isomorphism σ from Σ onto Σ' satisfy certain constraints, such as mapping given sort or operation names to themselves.
[9]The operations of *Ordered_Set_Type* are based on the dynamic-set operations described in [9].

(7) *Maximum*(S): This operation returns the element of a nonempty set S with the greatest key value.

(8) *Predecessor*(S, x): If S contains x and an element whose key is less than x's, this operation returns the element of S with the greatest key less than x's, and it returns *NIL* otherwise.

(9) *Successor*(S, x): If S contains x and an element whose key is greater than x's, this operation returns the element of S with the least key greater than x's, and it returns *NIL* otherwise.

We denote this signature by Σ. Assume that implementations of the *Boolean*, *Element*, and *Key* types are known. ADT behavior sampling can be used to find an implementation of *Ordered_Set_Type*, as follows: The searcher specifies the signature Σ for use in identifying candidate implementations for *Ordered_Set_Type* and randomly chooses a sample $\mathscr{S}$ of operational access computations on *Ordered_Set_Type*, such as those shown in Figure 3. The searcher provides a program schema P that describes, using terms over Σ, an algorithm to evaluate the results of the computations in $\mathscr{S}$. For example, this algorithm might check that

$$Empty(Newset) = true$$

and that if term x_2 has key k then

$$Search(Insert(Delete(Insert(Insert(Insert(Newset, x_1), x_2), x_3), x_1), x_4), k) = x_2.$$

Each admissible signature isomorphism $\sigma: \Sigma \to \Sigma'$, where Σ' is the signature of an ADT implementation I in the library, is used to create a program $\sigma^*(P)$ by instantiating P with σ. This program is executed using the implementation I, and if the program determines that I behaves acceptably, then I and σ are identified to the searcher for inspection.

7. CONCLUSION

We have presented and evaluated a new method, called behavior sampling, for automatically retrieving reusable components from a software library. Basic behavior sampling identifies components whose interface is compatible with a specification, executes them on a representative sample of inputs, and retrieves those components whose output matches output provided by the searcher. The mechanics of basic behavior sampling have been described, and the method's probabilistic basis has been established. The results of an experiment to assess basic behavior sampling has been presented. The results suggest that basic behavior sampling is precise (retrieves mostly relevant components) when used with small input samples and that it can be implemented efficiently. To enhance the flexibility and recall of basic behavior sampling, extensions have been considered. One extension permits the searcher to program arbitrary acceptance tests for output. Another extension allows abstract data type implementations and object classes to be retrieved with behavior sampling.

$Empty(Newset)$

$Empty(Delete(Insert(Insert(Insert(Insert(Newset, x_1), x_2), x_3), x_4), x_3))$

$Maximum(Delete(Insert(Delete(Insert(Insert(Newset, x_1), x_2), x_2), x_3), x_1))$

$Predecessor(Insert(Insert(Insert(Insert(Insert(Newset, x_1), x_2), x_3), x_4), x_5), x_4)$

$Search(Insert(Delete(Insert(Insert(Insert(Newset, x_1), x_2), x_3), x_1), x_4), k)$

$Successor(Insert(Insert(Newset, x_1), x_2), Minimum(Insert(Insert(Newset, x_1), x_2)))$

Fig. 3. Access computations on *Ordered_Set_Type*.

Significant additional research is needed to demonstrate the utility of behavior sampling and to exploit its potential. Although our experimental results suggest that basic behavior sampling exhibits good precision and efficiency, they are not conclusive. Further experimentation is necessary to confirm our results, to assess the recall of behavior sampling, and to address the extensions to basic behavior sampling discussed in Section 6. The issue of how to implement behavior sampling most efficiently requires thorough investigation. Finally, the question of how best to integrate behavior sampling with other retrieval mechanisms merits consideration.

Even the extensions we propose will not make behavior sampling universally applicable. Indeed, behavior sampling may be ill-suited for retrieving components with very complex interfaces, for example, graphical user interfaces. Since such components tend to be large and fewer in number than simpler components, more conventional means of retrieval, such as catalogs, may be adequate for locating them. It may be desirable to require that components of libraries to be searched with behavior sampling have interfaces that follow conventions facilitating target-interface specification. We are investigating what form such conventions might take.

REFERENCES

1. AGRESTI, W. W., AND MCGARRY, F. E. The Minnowbrook workshop on software reuse: A summary report. In *Software Reuse: Emerging Technology*, W. Tracz, Ed. IEEE Computer Society Press, Washington, D.C., 1988, 33–40.
2. AIKEN, M. W. Using artificial intelligence based system simulation in software reusability. *ACM SIGSOFT Softw. Eng. Notes 15*, 5 (Oct. 1990), 23–27.
3. ALLEN, B. P., AND LEE, S. D. A knowledge-based environment for the development of software parts composition systems. In *Proceedıngs of the 11th International Conference on Software Engineerıng* (May 15–18, 1989). IEEE Computer Society Press, Washington, D.C., 1989, 104–112.
4. ARNOLD, S. P., AND STEPOWAY, S. L. The REUSE system: Cataloging and retrieval of reusable software. In *Proceedings of the 1987 Spring Joint Computer Conference* (San Francisco, Feb. 23–27, 1987). IEEE, New York, 376–379.
5. BIGGERSTAFF, T. J., AND RICHTER, C. Reusability framework, assessment, and directions. *IEEE Softw. 4*, 2 (Mar. 1987), 41–49.
6. BOEHM, B. W. Improving software productivity. *IEEE Computer 20*, 9 (Sept. 1987), 43–57.

7. Burton, B. A., Aragon, R. W , Bailey, S. A., Koehler, K. D , and Mayes, L. A The reusable software library. *IEEE Softw.* (July 1987), 25–33.
8. Caldiera, G., and Basili, V R. Identifying and qualifying reusable software components. *IEEE Computer 24*, 2 (Feb. 1991), 61–70.
9. Cormen, T. H., Leiserson, C. E., and Rivest, R. L. *Introduction to Algorithms*. MIT Press, Cambridge, Mass., 1990.
10. Curtis, B. Cognitive issues in reusing software artifacts In *Software Reusability Volume II: Applications and Experience*, T. J. Biggerstaff and A. J. Perlis, Eds ACM Press, New York, 1989, 269–287.
11. Embley, D. W., and Woodfield, S. N. A knowledge structure for reusing abstract data types. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, Calif., Mar. 30–Apr. 12, 1987). ACM, New York, 360–368.
12 Fischer, G., Henninger, S., and Redmiles, D. Cognitive tools for locating and comprehending software objects for reuse. In *Proceedings of the 13th International Conference on Software Engineering* (May 13–17, 1991). IEEE Computer Society Press, Washington, D.C., 1991, 318–328.
13. Frakes, W. B., and Nejmeh, B A An information system for software reuse In *Proceedings of the 10th Minnowbrook Workshop on Software Reuse* (Syracuse, N.Y., July 28–31, 1987). Syracuse University.
14. Freeman, P. Reusable software engineering: Concepts and research directions In *ITT Proceedings of the Workshop on Reusable Software*. (ITT, Newport, R I., 1983).
15. Furnas, G. W., Landauer, T. K., Gomez, L. M., and Dumais, S. T. The vocabulary problem in human–system communications. *Commun. ACM 30*, 11 (Nov 1987), 964–971.
16 Goguen, J. A. Reusing and interconnecting software components *IEEE Computer* (Feb 1986), 16–28.
17. Goguen, J. A., Thatcher, J. W., Wagner, E. G., and Wright, J. B. Abstract data types as initial algebras and the correctness of data representations In *Proceedings of the Conference on Computer Graphics, Pattern Recognition, and Data Structures*. 1975, 89–93.
18. Guttag, J. Abstract data types and the development of data structures. *Commun. ACM 20*, 6 (June 1977), 396–404.
19. Hall, P. A Software components and reuse—Getting more out of your code. *IEEE Softw.* (July 1987), 38–43.
20. Honiden, S., Sueda, N., Hoshi, A., Uchihira, N., and Mikame, K. Software prototyping with reusable components. *J Inf. Process. 9*, 3 (1986), 123–129.
21. Horowitz, E., and Munson, J. B. An expansive view of software reuse. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept. 1984), 477–487.
22 Jones, T. C. Reusability in programming: A survey of the state of the art. *IEEE Trans. Softw. Eng SE-10*, 5 (Sept. 1984), 488–494.
23. Katz, S., Richter, C. A., and The, K. Paris: A system for reusing partially interpreted schemas. In *Proceedings of the 9th International Conference on Software Engineering* (Monterey, Calif., Mar. 30–Apr. 2, 1987). ACM, New York, 1987.
24. Lenz, M., Schmid, H. A., and Wolf, P. F. Software reuse through building blocks. *IEEE Softw.* (July 1987), 34–42.
25. Liskov, B., and Zilles, S Programming with abstract data types. *ACM SIGPLAN Not. 9*, 4 (Apr. 1974), 50–59
26. Lubars, M. D. Code reusability in the large versus code reusability in the small. In *Software Reuse: Emerging Technology*, W. Tracz, Ed. IEEE Computer Society Press, Washington, D C., 1988, 68–76.
27. Maarek, Y. S., Berry, D. M., and Kaiser, G. E. An information retrieval approach for automatically constructing software libraries. *IEEE Trans Softw. Eng. 17*, 8 (Aug 1991), 800–813.
28. McIlroy, M. D. Mass produced software components In *Proceedings of the 1969 NATO Conference on Software Engineering*. 1969, 88–98.
29. Onuegbe, E. Software classification as an aid to reuse: Initial use as part of a rapid prototyping system. In *Proceedings of the 20th Annual Hawaii International Conference on System Sciences* (Jan. 7–10, 1987) Western Periodicals, San Francisco, 1987, 521–529.

30. PARNAS, D. L. On the criteria to be used in decomposing systems into modules. *Commun. ACM 15*, 12 (Dec. 1972), 1053–1058.
31. PODGURSKI, A. AND PIERCE, L. Behavior sampling: A technique for automated retrieval of reusable components. In *Proceedings of the 14th International Conference on Software Engineering* (May 11–15, 1992). ACM, New York, 1992, 349–360.
32. PRIETO-DÍAZ, R. Implementing faceted classification for software reuse. In *Proceedings of the 12th International Conference on Software Engineering* (Mar. 26–30, 1990) IEEE Computer Society Press, Washington, D.C., 1990, 300–304.
33. PRIETO-DÍAZ, R., AND FREEMAN, P. Classifying software for reusability. *IEEE Softw. 4*, 1 (Jan. 1987), 6–16.
34. RUSSEL, G. Experiences implementing a reusable data structure component taxonomy. In *Proceedings of the 5th Annual Joint Conference on Ada Technology and Washington Ada Symposium* (Mar. 17–19, 1987). U.S. Army Communications-Electronics Command, Ft. Monmouth, N.J., 1987, 8–18.
35. SALTON, G. Another look at automatic text-retrieval systems. *Commun. ACM 29*, 7 (July 1986), 648–656.
36. STANDISH, T. A. An essay on software reuse. *IEEE Trans. Softw. Eng. SE-10*, 5 (Sept. 1987), 494–497.
37. TARUMI, H., AGUSA, K., AND OHNO, Y. A programming environment supporting reuse of object-oriented software. In *Proceedings of the 10th International Conference on Software Engineering* (Apr. 11–15, 1988). IEEE Computer Society Press, Washington, D.C., 1988, 265–273.
38. WEGNER, P. Capital-intensive software technology. *IEEE Softw. 1*, 3 (July 1984), 7–45.
39. WEYUKER, E. J. On testing non-testable programs. *Comput. J. 25*, 4 (Nov. 1982), 465–470.
40. WIRSING, M. Algebraic specification. In *Handbook of Theoretical Computer Science*, J. Van Leeuwen, Ed. MIT Press, Cambridge, Mass., 1990, 675–788.
41. WOOD, M., AND SOMMERVILLE, I. An information retrieval system for software components. *SIGIR Forum 22*, 3–4 (Spring–Summer 1988), 11–25.
42. WOS, L. *Automated Reasoning: 33 Basic Research Problems*. Prentice-Hall, Englewood Cliffs, N.J., 1988.

Received July 1992; revised and accepted February 1993