Support of Semantics Recovery during Code Scavenging using Repository Classification

Heinz Pozewaunig, Dominik Rauner-Reithmayer
Institut für Informatik-Systeme
Universität Klagenfurt
Klagenfurt, Austria
email: {hepo, dominik}@ifi.uni-klu.ac.at

Abstract

One of the hardest tasks to be fulfilled during the analysis of legacy systems is how to determine the precise semantics of program components. Investigating the internal data and control structures is difficult due to the huge number of possible implementation variants for the same problem.

To facilitate the task we propose to use components kept and described in a repository of reusable concepts as reference points. This becomes possible when behavior sampling is used as classification/retrieval strategy. In matching the results of isolated components from a legacy system against already executed components in a repository, one can tackle the problem of classifying legacy components without considering there internal structure. As a side effect, the population of the reuse repository is increased.

In this paper we propose a model to reuse the knowledge containd in a behavior based reuse repository for analyzing, classifying and understanding isolated executable components from a legacy system. Components not yet classfied will augment the repository.

Keywords: behavior sampling, component classification, reengineering, program understanding

1 MOTIVATION

One of the main topics in software reengineering is the task of analyzing parts of the system in order to (re)detect the functionality and meaning of such fragments. A broad range of methods for identifying, understanding, classifying, redocumenting, and reengineering program components in legacy systems has been proposed in the literature. Most of these methods can be applied very successfuly, if a-priori knowledge about program structure and programming style is available. As a consequence, if these assumptions do not hold, structure based analysis techniques must fail.

In contrast, to classify assets for the purpose of software reuse, extensive documentation is available to fulfill this task. But here the problem of correct interpretation of describing keywords arises. Since interpretation of keywords depends heavily on the cultural, social, and personal context [3, 1], successful classification depends on many human factors. To overcome such obstacles, researchers work on questions such as how to describe software without relying on human interpretation. Many approaches deal with formalizing the properties of component interfaces [6, 19] or using the inherent property of executabilty to directly determine meaning from software attributes.

Part of our motivation for the work described in this paper stems from a project to develop a reengineering tool [18]. Here, we were faced with the problem to perform semantics preserving code transformations. In quite a number of cases, this re-juvenation of code would be better performed by replacing parts of this code (a chunk) by some semantically equivalent component written according to state-of-the-art programming practices. Linking reengineering techniques with reuse experience seems promising in this situation. Thus, we were looking for domain specific components either as reference points or even as candidates for substitution of pieces of legacy code. As stated in [13] domain specific reusable components are considered as four times more valuable than general assets (abstract data-types, graphical user-interface functions, ...). Hidden in legacy systems, one can find a lot of high quality components that have been proven to be valuable during many years of operation. Since these components are required in one application, they are good candidates for further use in a similar domain and should be analyzed for potential reusability.

Subsuming we can say that on one hand there is the need for revealing semantics from legacy code in the field of reengineering and on the other hand there exist techniques (in the field of reuse) to describe components without (much) additional help from human experts. Our idea is to bring these two areas together with the aim of gaining benefit for both sides.

To bridge the gap between the demand for domain specific components and the task to detect unrevealed assets in the legacy code, we present a method for classifying and understanding program components.

This paper is organized as follows: In the next section we illustrate the difficulties in determining the meaning of components if analysis solely rest upon internal structure. In section 3 we present the structure of a reuse repository based on the behavior of reusable components. Furthermore, we sketch how a newly added component is classfied according to our approach. The detailed method is described in section 4. We reflect upon our work with a discussion in section 5 and conclude in section 6.

2 REVEALING PROGRAM SEMANTICS

This section sketches the problems structure based methods for program understanding are facing, in the case that no assumptions can be made about the analyzed components.

Program understanding is often viewed as the process of finding program plans in source code, which represent a certain meaning [15]. In taking a closer look to approaches in that field we see that most of them begin by analyzing some source-code instructions. The next step is to find out which program plans might use these instructions. Taking these plans as input, the program understanding methods try to infer higher level plans from that information [14].

The representation of plans, therefore, must contain details of the program structure and the relationship of source-code instructions with such plans. The representation varies from graphs [17], logical constraints [10] to abstract languages [9]. All these bottom-up approaches need to describe every possible implementation variant they want to recognize. This leads to either extensive plan libraries or to small and highly domain dependent plan libraries. But if we want to understand (parts of) programs for the purpose of reusing software, applying object oriented rearchitecting or redocumenting the dynamics of a system [16], the program structure itself does not matter: we only want to detect the semantics of programs.

We sketch the problem by a small example. Figure 1 shows two (semantically different) C functions `strfun1` and `strfun2` with identical signature and identical structure. The only difference one can find is in the calculation of the return value. This minor but important difference in the calculation of the return value discriminates between the semantics of the functions: The first of them performs a string

```
int strfun1 (const char *s1, const char *s2){
  for (;*s1 == *s2 && *s1 != '\0'; s1++,s2++);
  return (*s1 - *s2);
}

int strfun2 (const char *s1, const char *s2){
  for (;*s1 == *s2 && *s1 != '\0'; s1++,s2++);
  return (*s1 != '\0');
}
```

Figure 1: C functions performing an unknown task

comparison and returns a value of zero, if the two input strings are equal. The other one also performs a string comparison, but the return value is zero if the first input string `s1` is a prefix of the second input string `s2`. This difference is important if

1. a reuser searches for a function performing one of these two tasks or
2. a reengineer tries to determine the functionality of the respective component.

In a plan based approach it is costly in terms of time and money to describe all possible implementation variants (for the function `strfun1` some of them are shown in Fig. 2). In general the attempt to determine the whole variety of implementation variants is impossible.

Of course, these examples are small and simple, nevertheless they suffice to present the idea.

```
int strfun1_version1 (const char s1[],
                      const char s2[]){
  int i;
  for(i=0;s1[i] == s2[i] && s2[i] != '\0';i++) ;
  return (s1[i] - s2[i]);
}

int strfun1_version2 (const char *s1,
                      const char *s2){
  if(*s1 == *s2 && *s1 != '\0')
    return(strfun1_version3(++s1,++s2));
  else
    return(*s1 - *s2);
}

int strfun1_version3 (const char *a,
                      const char *b){
    while(*b != '\0' && !(*(a++) - *(b++))) ;
    return(*a - *b);
}
```

Figure 2: Semantically equivalent C functions with different structure

3 SELF DESCRIBING COMPONENTS

In this section a method for classifying components based solely on the property of executability of software is discussed. We extend this known technique to support automatic classification using test data as discriminative values.

3.1 Behavior Sampling

Podgurski, Pierce and Hall [12, 7] describe this approach to retrieve software components without the need of human interpretation. In general, if someone is looking for a certain component to be integrated into a software system, she/he is not interested in a particular structure or description, but in a special functionality. The "behavior sampling" technique is founded upon this observation and exploits the main difference of software artifacts to other information objects: their executability. Since every execution can be seen as an example of component behavior, the examples gathered in this way are considered as partial descriptions of the functionality performed by the component. It is obvious that a volatile execution eludes from direct observation for the purpose of behavior determination.

However, behavior manifests itself in data transformations. From an abstract point of view, every software component transforms some input into some output. If the library containing the software artifacts is organized in such a way that every artifact is directly executable, examples of behavior are generated on-the-fly by entering a "query". Thus, a query is a set of examples in the form of input-output tuples. Such tuples are carefully selected by the reuser searching for needed functionality (hereafter called the searcher).

The process of retrieving components by behavior sampling requires three interactive steps :

1. The searcher specifies the interface of the component.
2. The searcher chooses a small set of relevant input items and determines (eventually manually) the output.
3. The query (the set of input-output tuples) is entered into the retrieval system. The components stored in the software repository (which are in accordance to the interface specification) are executed on that input data.

In matching the computed output against the expected (eventually manually) calculated result values, components are included into the candidate set if there output matches, excluded if there is no match.

Since precision is in this case the domanant aspect, the searcher must

1. enter an sufficient number of examples and
2. the examples should reflect the main characteristics of the required functionality in the sense of domain tests.

On the one hand, if the cardinality of the query set is sufficiently large, only relevant components are selected. The result of a query is a component, resp. a (hopefully) small set of candidate components, matching the given input-output specifications. A candidate consists of the executable binaries, but also the source code and additional documentation are ingredients. In this way the searcher is allowed to check candidates to ensure the relevance of the component. On the other hand, if the query samples are chosen carefully, the cardinality of the candidate set is small enough to enable refinement of the results in a few iterative steps.

3.2 Behavior based classification

One of the main disadvantages of behavior sampling is the need to execute all components. To do this, an execution environment must be established which provides all necessary resources for executing the assets. This requirement cannot be fulfilled in every case, as some resources or preconditions cannot be maintained. Furthermore, executing components is time consuming and even with a high degree of parallel computation a moderate number of components can not be treated interactively.

In [11] the authors propose to use the history of executed program traces (test runs) as a knowledge base to describe the behavior of components. Starting from the presumption that all asserts in a software repository are tested carefully, these test cases are stored in the repository and serve as a (partial) description of the component. These "past pictures" of program runs are examples (partially) describing the behavior.

It is obvious that not all test cases are useful in the required sense. Test cases originally are designed with the intention to detect faults. But considering the theory of functional testing, black-box-tests aim to reveal the characteristics of components by checking domain boundaries [2]. In that sense domain test cases are a good choice to serve as discriminators between assets. We want to point out that an important property of test cases for the purpose of discrimination is that they are designed as *domain test data*. Such black box tests reflect the functionality per se in the most accurate way.

The component library (hereafter called component-behavior repository – CBR) contains signature abstractions, run time environments for executing components, test data and components. The notion component refers to a package including name, parameters, binaries and further comments and descriptions.

The structure of the CBR is coarse grained with respect to equivalent signatures. Signatures are generalized in the sense of the type view technique presented in [8] and are embodied as abstract signatures without implementation details. The library is divided into partitions containing components conforming to a generalized signature Σ_i. We will refert to such library partitions as Σ-partitions hereafter. The CBR also includes a run time environment for every Σ_i to support automated testing of components. Figure 3 depicts an overview of the CBR, where components C (together with their input-output tuples) are placed in segments i with respect to their signature Σ_i.

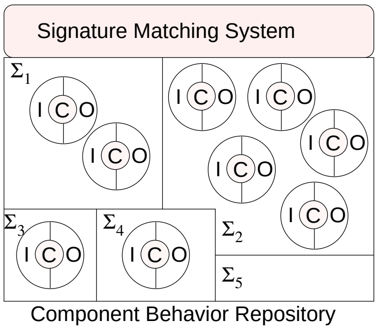

Figure 3: CBR Example

In figure 3 the conceptual architecture of the CBR is shown. To allow efficient access all test cases t_j of a signature Σ_i are administered in a set of test cases $\mathcal{T}$. As it is described in the following section we use test tuples as discriminators. To do so, we define a *component retrieval function cr* : $\Pi(\mathcal{T}) \rightarrow \Pi(\mathcal{C})$, which allows to map a subset of test-tuples to a relating subset of components $c \in \mathcal{C}$.

The CBR index is organized as classification tree, based on the test tuples as decision criterion. The main idea is to start from the set of all components of a Σ-partition and to add incrementally discriminating test data. Every added test tuple helps to generate a subset of components providing the output on the given input of the test tuple. The test tuple is chosen in the way that at least one component is extracted. A good discriminating test tuple is given, if the number of refined components is near to the number of the remaining ones.

If no tuple which helps to discriminate between the components can be found, the repository administrator must provide new test data according to the domain boundaries.

3.3 Classification example

The following example for inserting a new component into the CBR should help to understand the classification process.

In the repository a segment according to the signature `char` $\rightarrow$ `bool` contains ten (slightly modified ANSI-C) functions implementing predicates which determine certain properties of characters:

```
1  bool isalnum(char c);    7  bool islower(char c);
2  bool isalpha(char c);    8  bool isupper(char c);
3  bool iscntrl(char c);    9  bool ispunct(char c);
4  bool isdigit(char c);   10  bool isspace(char c);
5  bool isgraph(char c);
6  bool isprint(char c);
```

The CBR is indexed by a classification tree shown in figure 4. A node *ni* in the tree is poulated by components which demonstrate equivalent behavior on executing them on the test data given by the path from the root to the current node *ni*. Thus, an edge between two nodes is labeled with an input and branches to several output edges. A successing node contains only those components of the previous node, which behave in accordance to the input-output labeling the edge from predecessor to successor. More details on how to build the classification tree can be found in [11].

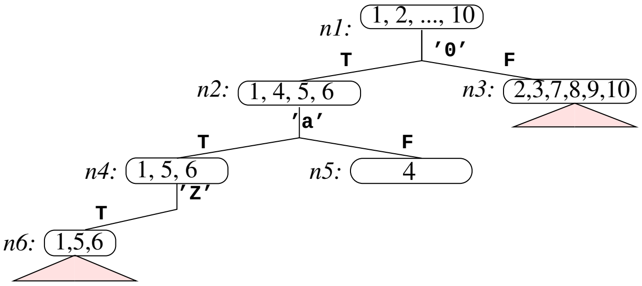

Figure 4: CBR classification structure

For the sake of demonstration, let us assume that we want to add a function `11 isxdigit : char` $\rightarrow$ `bool` to the library which checks if a given character is a hexadecimal digit character. The classification algorithm performs in the following manner: Starting at the root node, the component is inserted into node *n1*. To determine the correct successor node, the component is executed on the input '0' and after recognizing the calculated output `T`, the successor *n2* is chosen, since the path from *n1* to *n2* is labeled correctly with `('0', T)`. Node *n2* is not a leaf and, therefore, component 11 is executed with input value `'a'`. As the output is `T`, node *n4* is selected as successor. Now we execute 11 with input `'Z'` and because no edge is labled with the result `('Z', F)` we insert a new leaf node *n7*, link it to *n4* with a correctly labeled edge and add component 11 to the new leaf.

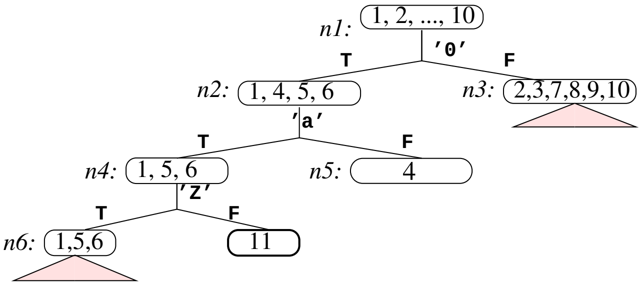

Figure 5: Classification structure after inserting `isxdigit` (11)

The result of the classification process is depicted in the classification tree of figure 5.

4 CLASSIFYING LEGACY COMPONENTS

In this section we want to show how to apply the software classification technique just presented to support reveal the semantics of legacy components. Since not every identifiable part is suited to be handled by our approach we discuss some preconditions legacy parts must fulfill. Subsequently, the process of scavenging and classifying components is described in detail.

4.1 Preconditions

If we want to find classifiable components in legacy systems we have to carve out meaningful, valuable and executable program fragments which fulfill the following criteria:

- The fragments should have meaning outside their original context. If components have no meaning no knowledge is available in the repository. This cannot be determined in advance in all cases.
- The fragments have to be executable outside their original context. This is a necessary precondition to start the matching process, since otherwise no path in the classification tree can be found.

The first criterion assures a certain degree of generality of the fragment, but cannot be proven during extraction. In program understanding the same problem is stated as chunk extraction. In this work we consider classifyable components as chunks and refer primarily to the definition and extraction technique for chunks given in [4]. The key issue of that definition is that a chunk has a coherent purpose and can be understood outside the original context.

Besides, we also view a chunk as a component (e.g. C-function) that can be executed outside its original context. A chunk should for this purpose be described by means of its input/output parameters (signature) and its global dependencies. With the term global dependency we mean the used (called) functions, not primarily defined within the respective chunk. This includes all global functions which are reachable in the call graph of the chunk. Hence, to isolate a program chunk we have to carve out its definition and all global dependencies.

Chunk extraction includes a certain amount of uncertainty. A fragment of legacy code can contain more than one chunk, since an executable part can be "shifted" through a set of source instructions. Likewise, considering that boundaries of chunks are not necessarily procedure boundaries, inclusion or exclusion of some statements at the boundary of a chunck might be improtant to obtain semantic coherence [5] Therefore, the determination of the correct signature for a chunk involves some fuzzyness, although, the referenced variables in that chunk give some good hints to build a signature. This uncertainty should be resolved in iterating through the process described in section 4.2. The process should be applied to every reasonable signature to increase the rate of recall, and in doing so increase the possibilty of detecting semantics.

4.2 Classification process

This section describes the process of analyzing legacy code and using the CBR as knowlege repository for determining the meaning of a program chunk. As additional benefit of this approach, not yet know chunks are identified as suitable candidates for adding them as reusable parts to the CBR.

The process contains several steps:

1. Extract and isolate a program chunk.
2. Generate a signature according to the used variables in the extracted chunk. If the chunk consists of a complete procedure, this step is trivial. If it is carved out of some larger monolithic code, the signature is to be determined by a dataflow analysis from all uninitialized variables to all defined variables spanning the state space of the chunk. (With prior knowledge, this "output set" can be reduced).
3. Determine the Σ-partition (together with the classification tree) for that chunk acccording to the signature.
4. Determine if the current node is a leaf.
   If it is a leaf we know that the chunk behaves (at least up to now) identically to all components in the current node and we continue with step 7a. Otherwise, if it is not a leaf node, select the input discriminating between the successor nodes and continue as discussed in section 3.3.
5. Execute the isolated program chunk on the selected input-data.
   To do so, we have to bed the chunk into an appropriate run time environment where it can be executed on the input data. The computed output is collected and stored for the next step. The run time environment is provided as part of the CBR Σ-partition description (see section 3 for details).
6. Match the computed input/output pairs against the appropriate input/output pairs specified in the classification tree.
   The input/output pairs computed in the previous step have to be matched against all "labels" of successing edges in the classification tree.

   The following results may be obtained by the matching procedure:

   (a) The calculated output matches perfectly with one edge output-label. This implies that the CBR contains more refined components demonstrating this behavior. We select the appropriate successor node and proceed with step 4.

(b) The calculated output matches with no label. Thus, no component in the CBR demonstrates the same behavior as the chunk. In that case we continue the classification process with step 7b.

7. (a) The chunk behaves in the same way as components in a leaf node.
From that we know that the chunk performs the same task as all other components in that leaf node and we have found at least a subset of the semantics of the chunk.

It is in the responsibility of a human expert to investigate further, whether the chunk in that leaf can be specialized and so up to now only a part of the semantics has been revealed. If this is the case the expert must provide new input data not yet on the classification path and test the chunk in order to reveal specialized behavior.

As a consequence, this chunk is considered as a new candidate for reuse purposes. In any case the expert is able to attach a significant amount of meaning to the analyzed legacy part.

(b) The chunk behaves equally to components in an inner node, but does not show the behavior of any components in the classification subtree of that node.
The chunk can be automatically classified as described in section 3.3. In that case we know that a new component has been found and the reuse benefit is obvious. But as the chunk has been derived from a legacy system, it has to be analyzed by a human expert and then amended by a correct description. The expert is supported by the information obtained by the classification process, such as the calculated input-output pairs and the available documentation of components showing partially the same behavior. So on the other hand program understanding for that chunk has also been obtained.

In figure 6 the process of program understanding is shown as a sequence of the above mentioned single steps using the CBR as knowledge base and plan library.

4.3 Role of Human

Steps 7a and 7b of the algorithm just described mentioned that the process of semantics recovery is not completed when sifting down through the classification hierarchy of the CBR comes to a halt. Why is this the case?

We have to be aware that the data-tuples used for CBR classification are carefully designed according to the specifications of the components contained in the CBR. Hence, domain partitioning on the basis of the specification can assure that the data points used in the classification can really discriminate among components and thus, in a pointwise manner, describe their semantics. – For the legacy chunk, we do not have a specification and hence there is no way to claim that the data points that lead to the classification up to the point where sifting down the hierarchy fully define different domain partitions. They only show that the component behaves in the data-points characterizing the partitions of the component stored in the CBR identical to this component. Hence, at least these partitions exists. But there may be more refined partitions and the partition boundaries might be drawn slightly different.

To check this, one can directly rely on a human expert. This was pointed out as simple fall-back strategy in the description of the algorithm. In this case, the human analyst is helped by knowing that the component is "a kind of" component X if the search stopped at component X and that it behaves at least partially like X. Certainly, this is already a big step forward over just having the plain code, as it provides heuristic guidance for further human program understanding.

In more complex situations, one can benefit further from the CBR. Since the Σ-partitions contain components of identical signature, the overall input-/output space of all components contained in a Σ-partition is identical. Components in a given Σ-space differ only in their specific input-/output mapping. Hence, one can use components attached to the node where sifting down came to a halt as generators for test data. Thus, further random testing can be used to show, whether a component classified up to a leave node has the same semantics as this node, or whether it is more refined. If it is more refined or if sifting down halted at an inner node, these random (or, specifically with inner nodes, strategically selected) test tuples can help to give further clues about the actual semantics of the component. The interpretation of these further experiments, of course, rests really with the human expert.

5 DISCUSSION

As one can see from the arguments in section 2 it is nearly impossible to infer semantic difference from the structure of different functions.

Returning to the demonstrative example: If we want to classify the first function of figure 1 (under the precondition that the CBR contains an ANSI-C compliant *string compare component*), we will see that `strfun1` behaves identical to our to examined string compare and this is also true for all variants shown in figure 2.

After applying the classification strategy presented above we can now easily determine that these functions belong to

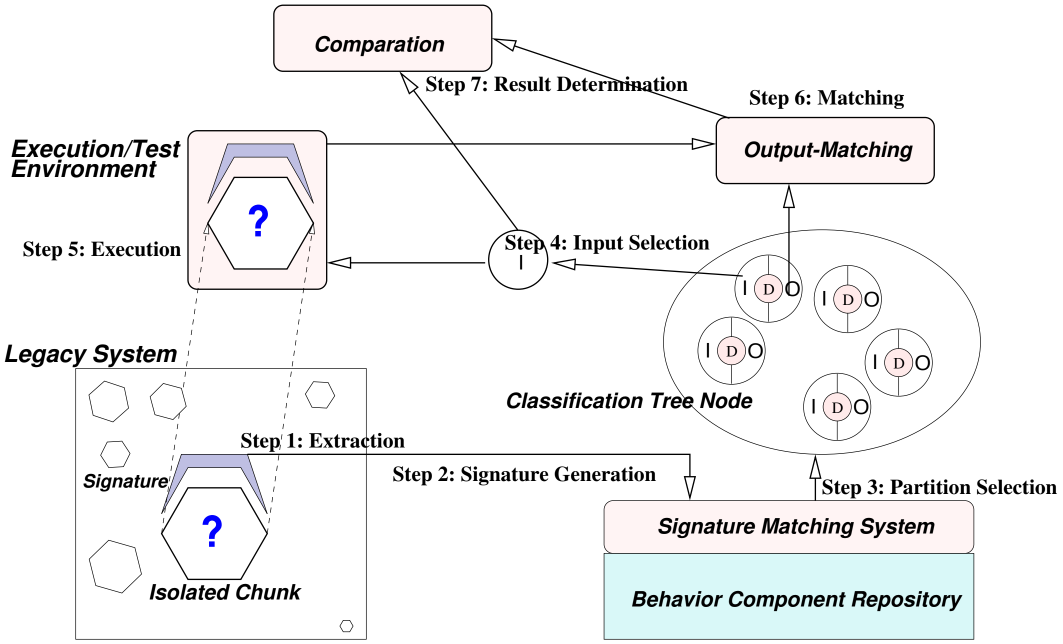

Figure 6: Classification Steps

the class of *string compare components*. This does not help to increase the population of our CBR, but we can state that the components in the CBR are of relevance within the application domain. Non-functional reasons might nevertheless be a cause for including the chunk just analyzed.

On the other hand we can use the information obtained about these chunks to analyze their quality and to improve the understanding of the legacy system in general. Due to the results of the classification, we can consider the process of behavior based classification as an alternative technique for program understanding. Of course, with that approach we can only understand chunks that are isolateable and when there exists a functionally equivalent counterpart in the CBR. But if these preconditions hold, we get high quality documentation for the analyzed part of the legacy system.

Furthermore, the knowlege in the CBR is manifested in executable, valuable software components of certified quality. Beyond the stated advantages the approach is useful for renovating legacy system. To increase the quality of the legacy system and to go stepwise into the direction of rearchitecturing, equivalent and modern components are first class candidates for substituting outdated legacy parts. If the behavior is exactly the same we may be confident about a semantics preserving transformation irrespective of structural differences.

If the precondition of an equivalent counterpart does not hold, the benefit is not in the field of program understanding, but in reuse! For example, if there is no counterpart of `strfun2` in the CBR, the classification process generates a new leaf node in the classification tree, indicating that the function may be important in this or similar application domains. As this function has already been proved valuable for a legacy system, the effort to engage a human expert for further analysis seems justified. The human expert is now in duty to document the specialization and to determine the quality of the component to ensure the high quality of components in the CBR.

Although classification based solely on behavior is very promising, some deficiencies are obvious. Side effects of components such as deletion of directories or files or writing to devices cannot be handled correctly on the tuple level. Also components requiring interactions with the enviroment cause difficulties in classifying and analyzing.

As a possibility to solve this problem, we investigate abstract signatures containing more information about dependencies to external interaction agents, like files, devices, and humans.

6 CONCLUSION

We presented an approach to understand, redocument and classify program fragments scavenged from legacy systems applying gathered knowledge incorporated in a reuse repository. The successful employment of a reuse technique in the field of reengineering brings a twofold benefit to normally non-overlapping areas: software reuse and program understanding. Since we apply the reuse repository on both fields, the return on investment justifies the effort of building a high quality reuse repository system.

References

[1] Steven Atkinson. Cognitive Deficiencies in Software Library Design. In *Proceedings of Asia-Pacific Software Engineering Conference and International Computer Science Conference*, pages 354–363. IEEE Computer Society Press, December 1997.

[2] Boris Beizer. *Software Testing Techniques*. Van Nostrand Reinhold – New York, 1990.

[3] Nik Boyd. Using Natural Language in Software Development. *The Journal of Object-Oriented Programming – JOOP*, 11(9):45–55, February 1999.

[4] Ilene Burnstein, Abdul Mirza, Katherine Roberson, Floyd Saner, and Abdallah Roberson. Knowledge engineering for automated program recognition and fault localization. In *Proceedings of the* 8^{th} *International Conference on Software Engineering and Knowledge Engineering SEKE' 96*, pages 85–91, Lake Tahoe (Nevada), June 1996.

[5] Ilene Burnstein and Floyd Saner. Applying Fuzzy Reasoning to the Program Understanding Problem. In *The* 10^{th} *International Conference on Software Engineering and Knowledge Engineering – SEKE'98*, pages 394–401, San Francisco Bay, California, June 1998.

[6] Yonghao Chen and Betty H. C. Cheng. Formalizing and Automating Component Reuse. In *Proceedings of* 9^{th} *International Conference on Tools with Artificial Intelligence – TAI 97*, pages 94 – 101, Newport Beach, California, November 1997.

[7] R. J. Hall. Generalized Behaviour-based Retrieval. In *International Conference on Software Engineering – ICSE93*, Baltimore, MD, May 1993. IEEE Computer Society, IEEE Computer Society Press.

[8] Gordon S. Novak Jr. Software Reuse by Specialization of Generic Procedures through Views. *IEEE Transactions On Software Engineering*, 23(7):401 – 417, July 1997.

[9] K. A. Konntogiannis, R. Demori, M. Galler, and M. Bernstein. Pattern matching for clone and concept detection. *Automated Software Engineering*, 3:77–108, 1996.

[10] Wojtek Kozaczynski, Jim Q. Ning, and Andre Engberts. Program concept recognition and transformation. *IEEE Transactions on Software Engineering*, 18(12):1065–1075, December 1992.

[11] Roland T. Mittermeir and Heinz Pozewaunig. Classifying Components by Behavioral Abstraction. In Paul P. Wang, editor, *Proceedings of the 4th Joint Conference on Information Sciences –JCIS'98*, volume 3, pages 547–550, RTP, North Carolina, USA, October, 23-28 1998. Association for Intellgient Machinery.

[12] Andy Podgurski and Lynn Pierce. Retrieving Reusable Software by Sampling Behavior. *ACM Transactions on Software Engineering and Methodology*, 2(3):286 – 303, July 1993.

[13] Jeffrey S. Poulin. *Measuring Software Reuse – Principles, Practices, and Economic Models*. Addison-Wesley, Reading, Massachusetts, 1997.

[14] Alex Quilici. A memory-based approach to recognizing programming plans. *Communications of the ACM*, 37(5):85–93, May 1994.

[15] Alex Quilici, Quiang Yang, and Steven Wood. Applying plan recognition algorithms to program understanding. *Automated Software Engineering*, 5:347–372, 1998.

[16] Dominik Rauner-Reithmayer and Roland Mittermeir. Behavior abstraction to support reverse engineering. In *Proceedings of the* 10^{th} *International Conference on Software Engineering and Knowledge Engineering SEKE' 98*, San Francisco, USA, June 1998.

[17] Charles Rich and Linda M. Wills. Recognizing a program's design: A graph-parsing approach. *IEEE Software*, pages 82–89, January 1990.

[18] Mario Taschwer, Dominik Rauner-Reithmayer, and Roland T. Mittermeir. Generating Objects from C code – Features of the CORET Tool-Set. In 3^{rd} *European Conference on Software Maintenance and Reengineering – CSMR'99*, Amsterdam, Netherlands, March 1999. IEEE CS Press.

[19] Amy Moormann Zaremski and Jeannette M. Wing. Signature matching: a Tool for Using Software Libraries. *ACM Transactions on Software Engineering and Methodology*, 4(2):146 – 170, April 1995.