INTELLIGENT RETRIEVAL
Open Journal

Review

Automated Software Component Recovery for Reuse

Hafedh Mili*; Odile Marcotte; Anas Kabbaj

Departement de Mathe´matiques et d'Informatique, Universite´ du Que´bec a` Montre´al, Succursale Centre, ille Montre´al (Que´bec) H3C 3P8, Canada

***Corresponding author**
Hafedh Mili
De´partement de Mathe´matiques et d'Informatique, Universite´ du Que´bec a` Montre´al, Succursale Centre , ille Montre´al (Que´bec) H3C 3P8, Canada

Article information
Received: April 23rd, 2024; **Revised:** June 1st, 2024; **Accepted:** June 17th, 2024; **Published:** July 20th, 2024

Cite this article
Mili H, Marcotte O, Kabbaj A. Automated software component recovery for reuse. 2024; 2(2). doi: https://doi.org/10.70705/ppp.ir.2024.v02.i02.pp55-60

ABSTRACT

Our study focuses on creating techniques and tools for constructing using reusable software and investigating strategies for generating such software. The general process of building utilizing reusable components consists of three stages: First, looking for and getting reusable parts according to partial requirements; second, figuring out how good those parts are for reuse; and third, maybe, modifying those parts to fit the current challenge [3, 16, 19]. This research provides an overview of several component retrieval strategies that have been created or are being investigated within the framework of SoftClass, an experimental CASE tool created at the University of Que'bec in Montre'al to investigate software reuse concerns [12, 15]. There are two main types of methods covered in this paper. The first is "traditional" keyword-based retrieval algorithms, which try to find a single component that meets the user's needs. The second type is "component composers," which look for combinations of components that could meet the user's needs. Other researchers have shown success in accessing software components and other types of information using the keyword-based approaches mentioned in § 3. These methods have already been put into practice. The component composers mentioned in section 4 are still in the theoretical stage of development, while efforts to identify efficient implementations are ongoing. Finally, we go over some potential avenues for further studies.

Keywords
Software reuse; Information retrieval; Concept networks; Intelligent concept matching; Function realization.

INTRODUCTION

Many scholars believe that software reuse is a key component to raising the quality and productivity of software products and software development processes [4]. In addition to the plethora of technical hurdles, software reuse faces several other obstacles, such as management practices and organizational structures [1, 4, 22]. Integrating reuse chores into developers' workflow in an effective and efficient way is one such difficulty [3]. Developers typically use development processors like code generators or high-level specification language interpreters in the generative or reusable processor approach [7], while in the building blocks approach they use the product of previous software development efforts to build new ones (see, for example, [21]). Before resorting to analytical approaches, the building blocks approach mandates the completion of three reusable tasks, which alters the conventional, analytical, divide-and-conquer strategy for system definition and design: 1) finding and retrieving reusable components using partial specifications, 2) determining how valuable these components are for reuse, and, perhaps, 3) customizing these components to fit the problem's requirements [16, 19]. Computer assistance for searching for and retrieving software components is the primary topic of this article.

According to some academics, developers won't regard reuse as a feasible option until the cost of the knowledge-intensive stages stated before is lower than, or much lower than [23] compared to the cost of creating software from the beginning[3, 16]. To keep these stages cheap and to minimize user interaction, we think sophisticated computer help is vital. This research provides an overview of several component retrieval strategies that have been created or are being investigated within the framework of SoftClass, an experimental CASE tool created at the University of Que'bec in Montre'al to investigate software reuse concerns [12, 15]. In this article, we will go over two broad types of methods: First, there are "traditional" keyword-based retrieval algorithms that try to find a specific component that the user requires, and second, there are "component composers" that look for different combinations of components.

where the user's requirements might be met. The latter are used just in cases when the former do not fulfill the user's requirements. Other researchers have shown success in accessing software components and other types of information using the keyword-based approaches mentioned in § 3. These methods have already been put into practice. Theoretical efforts to discover efficient implementations of the component composers, as mentioned in §4, have not yet been put into practice. In part 5, we wrap off by talking about

potential avenues for future studies.

2. Soft Class: An OO CASE Tool for Software Reuse

The trial's primary product line, distributed management software, was the target of the SoftClass project, which sought to improve software reuse for this line of products. The project's development revolved on an experimental CASE tool, which was briefly known as SoftClass (short for "software classification") to reflect its original purpose: to serve as a repository for reusable components. An open and versatile CASE tool, SoftClass was the result of a lot of hard work. The phrase "software component" in SoftClass may mean various things depending on the approach used to create the software, but it can also mean distinct "incarnations" of software products at different points in their development. Additionally, with the exception of the graphical user interface and a few custom-tailored editors, the majority of SoftClass products handle data and operations transparently.

Here are a few things you may do with the tools provided by SoftClass: 1) repackaging current structured software to suit the representation model, 2) searching for reusable components, and 3) offering computer aid for customizing reusable components to meet new demands. Although not directly related to the CASE tool, the repackaging tool set facilitates the creation of files that may be imported into SoftClass [13]. The ability to modify reusable components is supported by SoftClass's ability to record and trace transformations and activities throughout development. It involves anticipating and acting upon how local changes to a reusable component's representation at one level of development will impact successive levels [15].

What we call "objects" in SoftClass are really representations of software components with 1) "global attributes" and 2) "internal structure" that define how they work. An example of an attribute would be "Application Domain" for a process-like component, "Author", "Creation Date", "Function" for a data component, and "semantic contents" for the former. For process-like components, they also serve to define the component's external interfaces, which include input/output variables/flows; for data objects, they specify protocols. Two components make up attribute "values": 1) a human-readable description and 2) keywords used by different retrieval methods (see to § 3). In a nutshell, 1) administrative data and 2) a description of the black box's functionality make up the global attributes of components. A component's internal structure is defined by the subcomponents that make up the component and any relations that are required to be true between them. Other processes and data items may serve as subcomponents for process-like components. The aggregated pieces of data objects are their subcomponents.

The High Level Query System (HLQS) is a process-like software component that is visually represented in Figure 1 utilizing the graphical editor of SoftClass. We can see the "boundary" of HLQS in the circle. Variables and flows that are input and output are represented by the squares on the circle. The HLQS subcomponents are represented by the parts within the circle. In this study, we will not go into additional detail about the interface's anatomy. Just to clarify, the window to the right of the HLQS display area has two editors: one for the object whose properties are being modified (the top editor in this example), and another for the component whose properties are being edited within HLQS (none in this case, the below editor). The attribute "Function" is chosen for HLQS (upper attribute editor). First, on the right side of the attributes list, under the label Values, you can see the whole set of keywords used to describe that attribute; in this example, there is only one item, "DataBaseInterrogation." two things: first, a brief description of the attribute (such as "HLQS is a high-level interface..."), which appears in the top part of the attribute editor. Three data flows and three processes make up HLQS's six subcomponents, as per SoftClass's concept of software components. All of the connections and relationships inside and between HLQS's constituent parts make up its internal structure. Relation objects [10], which are specific objects that stand in for relations, serve as an internal representation of the structure. Users of SoftClass don't have to manually alter these relations since they are automatically generated from the aforementioned graphical editor designs. But they're a part of the documentation of "structural transformations" now. Additionally, when they are created, a number of compatibility requirements are checked. For instance, these checks are done to ensure that the data that is really flowing into a (already built) process is compatible with the data that is expected of it.

Here we can see, in Figure 1, how the graphical component editor in SoftClass depicts the internal structure of a process-like component.

All of the software component descriptions in SoftClass are just instances of categories or description templates. We roughly have one category for each stage of development and one for each sort of component inside a specific development technique (e.g., process/procedure vs data). A (Smalltalk) class hierarchy is used to organize categories in a way that takes use of their commonalities. One set of relevant characteristics and their semantics (number of values, domain of keyword values, etc.) is described by each category. Another set of acceptable sub-component categories and generic internal relations are also part of each category.

3. Keyword-Based Retrieval Of Reusable Components

SoftClass includes a search tool-- called SoftSearch-- that enables developers to retrieve software components based on the keyword values of their attributes. We implemented three matching algorithms: 1) a weighted boolean retrieval (see e.g. [20]), 2) conceptual distance measures, and 3) classification. Both conceptual distance measurements and classification use semantic relationships between keywords— hierarchical relationships for the case of classification. We begin by discussing the structure of queries and the operations that can be performed on them. We conclude in § 3.4 by discussing some of the implementation choices we are currently facing.

3.1. A Multi-Faceted Classification of Components

Our "attributes" are equivalent to the "facets" or dimensions used in Prieto-Multi-Facetted Diaz's classification of software components [17] for the sake of retrieval. If an attribute, like "Function," may have more than one value, such values are not partial values but alternative values. We specified a lexicon of important terms and phrases for each property so that users may choose those values. Parts of the phrase generalization hierarchies for the "Application Domain" (Figure 2.a) and "Function" (Figure 2.b) attributes are shown in Figure 2.

The majority of the time, the keywords associated with an attribute are structured in a single hierarchy, with the attribute name serving as the root. You may have noticed that the "Application Domain" keyword hierarchy takes its cues from the categorization structure of the (ACM) Computing Reviews [2]. Information retrieval professionals sometimes use the term "Broader-Term" to describe the connection between important terms, which is a loose kind of generalization [20].

3.2. Queries

Our choice for the representation of queries involved a trade-off between flexibility and expressiveness, on the one hand, and allowing users to specify the most common queries most easily and most efficiently, on the other. As a guiding principle, we likened the specification of a query to the process of specifying a prototypical component. Accordingly, the simplest form of a query is a list of so called attribute query terms (AQTs), considered to be ANDed. In its simplest form, an AQT consists of an attribute, and a list of key phrases, considered to be ORed. In the actual implementation, each AQT is assigned a weight and cut-off point, used for weighted boolean retrieval, dis- cussed further below, and conceptual distance measures (see § 4.2), respectively. Symbolically:

- Query ::= AQT □ AQT AND Query
- AQT ::= Attribute Weight CutOff ListOfKeyPhrases
- ListOfKeyPhrases ::= KeyPhrase □ KeyPhrase OR ListOfKeyPhrases

A single AQT retrieves the components whose attribute <Attribute> has at least one value in common with <Lis- tOfKeyPhrases>. Viewing attributes as functions, an AQT denoted by the four-tuple <Attribute, Weight, Cut Off, Lis- tOfKeyPhrases> retrieves the components C such that Attribute(C) □ ListOfKeyPhrases □ □. The query denoted by the tuple (AQT1,..., AQTk), returns the intersection of sets of components that would have been returned by the indivi- dual AQTs. With weighted boolean retrieval, components are assigned numerical scores computed as follows:

k

In SoftSearch, the same query (i.e. list of AQTs) may be matched against the repository of reusable components using any one of the three search algorithms, yielding different— possibly disjoint— sets of components. Hence, we associate with each query three answer lists corresponding to the three search types. Further, we associate to each search type a search parameter used to filter the corresponding list of answers. With weighted boolean retrieval, a cut- off variable (between 0 and 1) indicates the minimum score that a component must achieve to be included in the list of answers. With conceptual distance measurement and classification, the cut-off value refers to the maximum number of elements to be included in the answer list.

3.3. Operations On Queries

Queries in SoftSearch may be combined, once processed/searched, using the usual set-theoretic operators (union, intersection, and set difference) directly on the answer lists. Although the terms of a query (AQTs) are implicitly ANDed, operations on queries enable developers to formulate arbitrarily complex logical expressions of AQTs. Trivi- ally, we can express any logical expression by operations on single AQT queries. for a number of reasons, both practi- cal and theoretical, we chose not to generalize the format of queries so that they can handle alternation and negation directly. First, we believe that such queries are relatively uncommon so as not to warrant complicating the format of queries, or offering two different formats. For example, negation is often used in information retrieval systems to refine an initial query that returned too many irrelevant items that share some common property. Second, from a theoretical point of view, negation is difficult to express mathematically in the context of graph-based conceptual dis- tance measurements (see [18]). Further, semantically, classification is hard to interpret if a query contains two ORed AQTs that correspond to different attributes [9].

3.4. Conceptual Distance Measures

Conceptual distance measures extend boolean and weighted boolean retrieval by replacing the match/mismatch relation between attribute query terms (AQTs) and the corresponding component attributes, by measures of conceptual similarity between their respective keyword values. We use path lengths between keywords within keyword taxo- nomies and conceptual hierarchies (see § 3.1 and Figure 2). Let x and y be two nodes from a connected undirected graph. We define distance(x,y) as the shortest path length between x and y. The reader can easily check that distance satisfies the properties of a metric, i.e.:

1) distance (x,x) = 0, zero property,
2) distance(x,y) = distance(y,x), symmetric property,
3) distance (x,y) □ 0, positive property, and
4) distance(x,y) + distance(y,z) □ distance (x,z), triangular inequality.

Referring to the example of Figure 2, distance(Forecasting, QuestionAnswering) = 3, corresponding to the path6 <Forecasting, DecisionSupport, InformationRetrieval, QuestionAnswering>, and distance (InformationRetrieval, Ques- tionAnswering) = 1, corresponding to the single edge <InformationRetrieval, QuestionAnswering). While neither number means much in terms of conceptual "distance", experiments showed that the resulting rankings of pairs of con- cepts are consistent with those made by human subjects [18].

In other work, we used distance to define a measure of conceptual distance between sets of nodes within a graph [18] that satified the metric property and that correlated with human relevance assessments in document retrieval experiments [18]. For component retrieval, we developed a different measure for two reasons: 1) DISTANCE suf- fered from non-intuitive mathematical irregularities7, and 2) a different interpretation of multiple values— namely, alternative, rather than partial values. Let MEASURE be SoftSearch's measure of conceptual distance between sets of values. Notwithstanding cut-off points (see below), MEASURE can be defined as follows:

MEASURE(X,Y) □

$$\text{Min} \quad \text{distance}(x,y) \qquad (3)$$
$$x \square X,\ y \square Y$$

The reader can check that MEASURE is symmetric and satisfies the zero property. However, MEASURE does not satisfy triangular ine-

quality. This is not a problem as triangular inequality is only needed when retrieval uses "concep- tual distance clustering" of documents [18], which is not the case with SoftSearch. To simplify notation, we write MEASURE (AQT,C), where AQT =(<Attribute>, <weight>, <cut-off>, <list of values>) is an attribute query term, and C a component with applicable attribute <attribute>, to denote MEASURE (<list of values>,<list of values of <attri- bute> for component C>). MEASURE values are used to compute scores for components, with the highest score (1.0)

corresponding to a zero value for MEASURE. Components C such that MEASURE(<AQT>,C) is above the cut-off point, where <AQT> is an attribute query term, are assigned a zero score. Formally, let AQTi = (atti, wi, cuti, listi), we define SCOREi as follows:

1.4. Enumerating Realizations

Practically, a developer is not interested in knowing whether his needs can be met by using the existing library of components, but rather in how they can be met. We now turn to the problem of enumerating the possible realiza- tions of a function g. Intuitively, we understand that many realizations are essentially "equivalent". For instance, if function f is invoked at the kth stage of realization R but can also be invoked at the previous stage, R will not be altered in an essential way by "moving" f from the kth stage to the (k□ 1)rst stage. On the other hand, R may contain redundant functions, i.e., functions f such that R\□ f□ is also a realization. Thus in order to decide whether two realiza- tions are equivalent or not, we have to normalize them. To this end, we make sure that a function is invoked at the earliest possible time, i.e., as soon as all its inputs become available. Let T0(R) denote Input (g), and Tk(R) denote

k

Input(g) □ (□ Output(Fp)), for k = 1, ..., s. Thus Tk(R) is the set of inputs that can be used at the (k□ 1)rst stage. We

p□ 1

say that realization R is in normal form if every function f of R is invoked at the stage k such that Input (f) is a subset of Tk□ 1 but not of Tk□ 2 (where T□ 1 □ □).

If R is not already in normal form, we can compute its normal form by calling Algorithm TryToRealize with the following values for the input parameters:

- C is the collection of functions that appear in R — call it C(R)—, and
- T is Ts(R), where s is the length of R.

Of course, in this case, the parameter Answer always takes the value "true", and the second output parameter is a realization of g in normal form that contains exactly the same functions as R. Thus in order to compare two realiza- tions, we may compute their normal forms and verify that these are identical. For obvious reasons, we want the

enumeration algorithm to produce realizations in normal form only. We also want realizations to be "minimal", at least in the sense that if a function is removed from the realization, then g cannot be realized in the same number of stages. We define minimal realizations as follows. We say that a collection of functions F covers U if Output (F) con- tains U. F is a minimal cover of U if for every f belonging to F, F - □ f□ does not cover U— where the minus ("-") sign denotes set difference. Finally, let R = □ Fk □ k = 1, 2, ..., s□ be a realization of g in normal form. We say that R is a minimal realization if and only if Fk is a minimal cover of (Input (Fk□ 1) □ Output (g) □ Output (Fk)) - Tk□ 1(R) for k = 1, 2, ..., s-1 and Fs is a minimal cover of (Output (g) □ Output(Fs)) - Ts□ 1(R). In other words, R is minimal if every function invoked at the kth stage is needed to produce a new type of Output (g), or a new input for the next stage.

Before enumerating minimal realizations in normal form, we restrict the collection of functions to be considered by the enumeration algorithm: clearly, a function may belong to a realization of g only if it also belongs to the trivial realization. On the other hand, such a function is required to produce a "useful" output. Let us define a directed graph G whose vertices are the functions of C, and where (f, h) is an arc if and only if Output (f) □ Input (h) □ □. Again, the function f may appear in a realization of G only if there exists a path from f to h in G for some h such that Output

(h) □ Output (g) □ □. Thus we replace the collection C by:

C0 □ □ f □ f belongs to the trivial realization□ □

□ f □ there exists a path from f to h in G for some h such that Output (h) □ Output (g) □ □□.

In the same vein, we shall use T0 instead of T, where T0 = Input (C0) □ Output (C0).

While replacing C by C0 and T by T0 does not reduce the theoretical complexity of the problem, it could sub- stantially reduce the time consumed by the algorithm, especially if the bank of functions is "sparse". The above pro- cedure is called at the kth stage, and it assumes that Fi (for i = 1, 2, ..., k-1) has already been chosen. It enumerates all minimal realizations in normal form that agree with the prefix □ Fi □ i = 1, 2, ..., k-1□. When EnumerateRealizations is invoked for the first time, its input parameters are k = 1, F0 = □ f0□, T□ 1 = □ and g, where f0 is a fictitious function such that Output (f0) = Input (g).

2. Directions

A lot of new directions are opening up for the research in this publication. Our efforts go in two ways when we use keyword-based techniques. First, from an engineering and development perspective, we are integrating search capabilities into SoftClass's editing tools in a "modeless" way. This will allow for the seamless integration of search capabilities with SoftClass's traditional, usually analytical, functionalities. Users experienced with competing search engines that had separate and unique user interfaces, for example, griped about the time wasted navigating between them [15]. In addition, we are in the process of compiling libraries of reusable software components in order to evaluate and compare the different approaches. Creating semi-automated techniques to add keywords to component properties is one component of these initiatives [15].

There is a great deal of unfinished business on the theoretical and technical levels in relation to bottom-up design assistance. The algorithms discussed in the preceding section have been implemented individually in LISP on an experimental basis until a number of theoretical and practical concerns are overcome. This is in contrast to the fully developed keyword-based approaches in SoftClass. There are two main concerns that we have right now. The operational descrip-

tion and characterization of "type"'s" are our primary concerns. For example, the significance of the realizations may increase or decrease based on how expressively the types are described, such as between types used at the programming level and types used in the application domain. The second point is that we don't have much control over the "quality" of the realizations, and the function realization issue is computationally prohibitive, as shown in sections 4.2 and 4.3. In an effort to better understand which realizations are "very interesting" and which are "utterly uninteresting," we are running the enumeration algorithm through a selection of UNIX's C libraries. Based on the results, we can hopefully refine the algorithm to generate more "interesting" realizations and discard the less interesting ones. The results quality is affected by both the type definition and type matching, which are not unconnected concerns. Lastly, from a development perspective, there are substantial technical problems in creating an effective implementation of the method Enumerate-Realizations—and variants thereof—using a relational database representation of software components. We anticipate that developers will be able to continue about their work uninterrupted only when the algorithm identifies "promising" options, since the search will run in a "background" mode dictated by response-time concerns.

REFERENCES

1. 1. With Brad J. Cox, in Addison Wesley's Object-Oriented Programming: An Evolutionary Approach, Readings, MA, 1987.

A new computing reviews classification scheme was proposed by Denning, Parker, Ralston, Reilly, Rosenberg, Walston, Willoughby, Sammet, and Blum in July 1981 in the Communications of the Association of Computing Machinery, volume 24, issue 7, pages 419-434.

Thirdly, in July 1987, Gerhard Fischer wrote an article for IEEE Software titled "Cognitive View of Reuse and Design" (pp. 60–72).

Section 4 of Peter Freeman's 1987 book, Tutorial: Software Reusability, covers "Reusable Software Engineering: Concepts and Research Directions" and appears on pages 10-23.

5. Computers and Intractability, by M. Garey and D. Johnson, published in 1979 by Freeman in San Francisco.

6. In 1978, Computer Science Press published Fundamentals of Computer Algorithms by E. Horowitz and S. Sahni.

7. "An Expansive View of Reusable Software" by Ellis Horowitz and John B. Munson; published in 1984 in the IEEE Transactions on Software Engineering, volume 10, issue 5, pages 477–487.

Eighth, "Fuzzy Relations" by Abraham Kandel, in Fuzzy Mathematical Techniques with Applications (Addison Wesley, 1986), pages 102–122.

9. Hafedh Mili, in The George Washington University's Building and Maintaining Hierarchical Semantic Nets, August 1988. Advanced Research Paper

10. in "An Object-Oriented Model Based on Relations," Hafedh Mili, John Sibert, and Yoav Intrator wrote in 1990 for the Journal of Systems and Software, volume 12, pages 139–155.

11. "Enhancing Software Reuse for Distributed Systems Management," Second Progress Report, TANDEM Computers & University of Quebec at Montreal, August 1990, by Hafedh Mili, Eric Fradette, and Manon Grenier.

Twelve. "SoftClass: An Object-Oriented Tool for Software Reuse" by Hafedh Mili, in Bertrand Meyer's Proceedings of TOOLS 5, Santa Barbara, CA: Prentice Hall, July 29–August 1, 1991, pp. 303–317.

15. "Managing Documentation for Software Reuse" by Hafedh Mili and Manon Grenier, published in 1992 in Information and Decision Technologies, volume 18, pages 115–134. 19.

14. "Intelligent Component Retrieval for Software Reuse" by Hafedh Mili, Odile Marcotte, and Anas Kabbaj was published in the proceedings of the Third Maghrebian Conference on Artificial Intelligence and Software Engineering (pp. 101-114), which took place in Rabat, Morocco, on April 11-14, 1994.

14. "Practitioner and SoftClass: A Comparative Study of Two Software Reuse Research Projects," in Journal of Systems and Software, vol. 27, May 1994, by Hafedh Mili, Roy Rada, Weigang Wang, Karl Strickland, Cornelia Boldyreff, Lene Olsen, Jan Witt, Jurgen Heger, Wolfgang Scherr, and Peter Elzer.

16. "Reusing Software: Issues and Research Directions" by Hafedh, Fatma, and Ali Mili Paper presented at the 1995 IEEE Conference on Software Engineering. To appear. Technical report 184 (April 1992), UQAM Department of Mathematics and Computer Science, is another source.

17. "Classifying Software for Reusability" by Ruben Prieto-Diaz and Peter Freeman was published in January 1987 in the IEEE Software journal, pages 6-16.

18. "Development and Application of a Metric on Semantic Nets" by Roy Rada, Hafedh Mili, Ellen Bicknell, and Maria Blettner was published in January/February 1989 in the IEEE Transactions on Systems, Man, and Cybernetics, volume 19, issue 1, pages 17-30.

19. "Support for Reusability in Genesis," published in the August 1988 issue of the IEEE Transactions on Software Engineering, was written by C. V. Ramamoorthy, Vijay Garg, and Atul Prakash.

Introduction to Modern Information Retrieval, by Gerard Salton and Michael McGill, New York: McGraw-Hill, 1983, p. 20.

Paper presented at the 1990 IEEE Computer Society Press conference on software reuse was written by Mark A. Simos and titled "The Domain-Oriented Software Life Cycle: Towards an Extended Process Model for Reusability." The conference was edited by Will Tracz and included the following sections: 21.

IEEE Press, 1987, pp. 358-363, "Software Reuse: Motivators and Inhibitors" by Will Tracz.

24. "Can Programmers Reuse Software?" by Scott N. Woodfield, David W. Embley, and Del T. Scott Software by IEEE, sections 52–59, July 1987. 2.

24. "Fuzzy Sets" by Lotfi A. Zadeh, published in 1965 in Information and Control, volume 8, pages 338–353.