Artificial Intelligence for Engineering Design, Analysis and Manufacturing (2001), *15*, 21–35. Printed in the USA.
Copyright © 2001 Cambridge University Press 0890-0604/01 $12.50

A case similarity metric for software reuse and design

PAULO GOMES AND CARLOS BENTO
Centro de Informática e Sistemas da Universidade de Coimbra, Coimbra, Portugal
(RECEIVED March 14, 2000; ACCEPTED June 20, 2000)

Abstract

When the idea of software reuse appeared in 1968, new horizons for software design were open. But some major problems appeared and most of the expectations were not met. One of the problems encountered is the selection of the right software component. This is related not only to the similarity between the desired functionality and the function delivered by the retrieved software component, but also to the effort needed to modify the chosen component to accommodate the desired functionality. Most of the research done in the case-based reasoning area has been in developing accurate and efficient retrieval algorithms. We think that case-based reasoning retrieval concepts and ideas can be successfully applied to software reuse. In this article we propose a metric to assess similarity between software cases supported on functional and behavioral knowledge. One important aspect of this metric is that reusability is taken into account to estimate the amount of effort needed to reuse retrieved software cases. We also present experimental work that shows that similarity at the functional level is the most important aspect of the similarity metric proposed.

Keywords: Case-Based Reasoning; Similarity Metric; Software Design; Software Reuse

1. INTRODUCTION

Software programming is a difficult design task, mainly because of the complexity involved in the process. Nowadays this complexity is increasing to levels in which reuse of previous software designs is very useful to shortcut the development time. The main idea of software reuse is to use previous software components to create new software programs (Prieto-Diaz, 1993). Thus software reuse is software design, where previous components are the building blocks for the generation of new systems. The idea of reusing software embodies several advantages. It improves productivity, maintainability, portability, and quality of software systems. One of the essential problems in software reuse is the retrieval and selection of suitable software components from a large library of components (Prieto-Diaz & Freeman, 1987).

Design is a cognitive task involving different types of knowledge and reasoning types, which implies a rich knowledge representation framework and a multireasoning system. This multiplicity of knowledge types and reasoning mechanisms makes difficult the development of a unifying methodology for design, and what generally works for software design does not work for architectural design. One possible way of overcoming this difficulty is to use "chunks" of solutions or subsolutions to problems in solving new design problems, thus reusing previous reasoning results. Case-based reasoning (CBR) is an area in artificial intelligence that deals with retrieval and reuse of old experiences to solve new problems (Kolodner 1993; Maher et al., 1995). Similar to software reuse, retrieval and ranking of the right experiences are important subjects to CBR. A great deal of work has been done concerning case similarity by the CBR community, and some attempts have already been made in using CBR concepts and ideas in the software reuse domain (Fouqué & Matwin 1993; Smyth & Keane 1995).

From the software reuse viewpoint, most of the time retrieved software components must be modified in order to deliver the required functional specifications (Prieto-Diaz & Freeman, 1987). This fact leads to a second important problem, not only in software reuse, but also in CBR. Reusability of software components must be assessed to determine which one is the easiest to adapt to accomplish the required functionalities.

Two main issues of our research are the similarity of software components and the degree of reusability of these components. Having in mind these issues, we propose a metric to assess the similarity between software component

Reprint requests to: Paulo Gomes, Departamento de Engenharia Informática, Polo II—Pinhal de Marrocos, 3030 Coimbra, Portugal. E-mail: pgomes@dei.uc.pt

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

functionalities and required functionalities. As part of this metric we also propose a metric for estimation of the reusability of software components. The similarity metric is based on functional and behavioral knowledge, where the actual behavior of the retrieved case is compared with the desired behavior for the new problem. The reusability metric uses behavior knowledge about the component's structure complexity to assess the effort needed to modify the retrieved function to deliver the new functional specifications. These metrics are used in our system—CREATOR II—a CBR system for software design.

In the next section we discuss some issues on the subject of similarity in software reuse. Section 3 presents CREATOR II architecture. Section 4 shows the case representation used in our system, which focuses on two representation levels: functional and behavioral. Section 5 presents the case similarity metric for software reuse. In this section we also present an example to illustrate some of the issues related to our metric. The next section presents and discusses some experimental work done with CREATOR II. Section 7 describes other works developed in this area, and compares them with our approach. Section 8 presents some conclusions.

2. ISSUES IN SOFTWARE REUSE SIMILARITY

In general, software components can be described by the function they perform (function), how they perform it (behavior), and how they are implemented (structure) (Prieto-Diaz & Freeman, 1987; Rosenman & Gero, 1998). A case representation for software design must be able to represent these three aspects. In our system, we use a function-behavior case representation (FBCR) formalism to describe software design. Within this formalism, functionality is represented as the function of the design, and case behavior describes how the software component achieves the function. Because the software program corresponds to the design structure, we also store the program code in a file, as part of the knowledge base. The next section explains FBCR in detail.

An important issue in choosing the best component for reusability is deciding which components are more easily adapted. Generally, good guidelines for predicting reusability are small size of code, simple structure, and good documentation (Prieto-Diaz & Freeman, 1987). Starting from the assumption that two functions have the same functionality, these three guidelines are used in our system to rank candidate functions for reuse.

Classifying functions is very important for accuracy and efficiency of a reuse system. Typically this task is performed by humans (Prieto-Diaz, 1991). In CREATOR II, function classification is done by the user and is supervised by the knowledge engineer responsible for the system.

To reuse old software components, the user must specify the functionality required for the new design. This functional description is made at an abstract level, and it comprises functional structure, function classification, function interface, and function output (when appropriate). The user can specify a hierarchy of functions describing the problem function structure. The function classification in CREATOR II is defined using a function taxonomy. In software design, the function interface is the set of parameters for the function. These parameters can be used to judge function similarity by classifying them along several attribute axes. Some of the attributes used in our system are input or output, the role of the parameter in the function behavior, and the data type of the parameter.

3. CREATOR II

CREATOR II is a CBR system in the domain of the design of digital circuits using the VSIC hardware description language (VHDL). The VHDL language is a procedural language like Pascal or C, used to specify the function and behavior of digital circuits. The main goal of our system is to reuse the VHDL programs used to design previous circuits in the design of new ones.

The architecture of CREATOR II is based on four different modules: VHDL to FBCR (which is the case description language used by CREATOR II) converter, knowledge base, CBR module, and FBCR to VHDL converter (see Fig. 1).

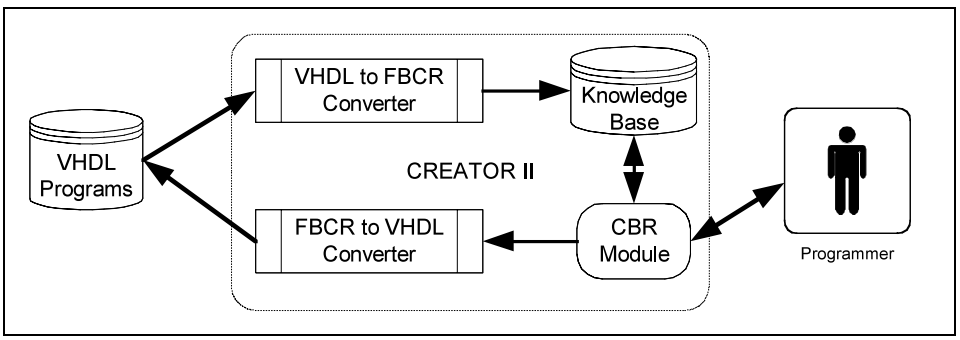


Fig. 1. The architecture of CREATOR II.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

The converter modules make the conversion between the VHDL language and the internal representation of CREATOR II. This conversion is fully automated, which speeds up the construction of the initial case base along with its maintenance.

The knowledge base comprises five different parts: index structure, case library, function class taxonomy, data class taxonomy, and the set of used adaptation plans.

Functions or procedures are the basic building blocks for procedural software programs. Thus, cases are indexed by their functions, allowing a case to be retrieved in several ways. This indexing scheme allows also the composition of different case pieces to create a new solution. The indexing structure can be divided into two parts: the function class level and the function instance level. The first part is the more abstract part of the indexing scheme, helping to provide a better problem specification and allowing function abstraction. It is also used for establishing similarity degrees between functions. The second level comprises the function instances, that is, the functions that are in the cases. The function class level is a hierarchy of function classes. Each node in this tree represents a function class, which is a general function specified in the knowledge base. An edge represents a-kind-of relations between two function classes. In the function instance level, nodes represent function instances; associated with each node is more information about the function. This information is used for retrieval and other operations, and includes the function parameters, the function behavior complexity, the number of lines of code, and the number of comment lines. The parameters comprise information about the classification of the parameters, that is, basically, if a parameter is input or output, and which data type it is. The behavior complexity, code lines, and comments are used to compute the function's reusability. There are three types of links: between an instance and a class (is-a arcs), between two instances (is-part-of arcs), and between an instance and a case (belongs arcs). A node can only have a function class and belong to a single case. Figure 2 shows an example of an index structure with one case.

The case library comprises the set of cases representing the VHDL programs, along with the VHDL program files associated with each case. Cases are represented using the FBCR formalism, which is described in the next section. This case description represents a case function as a functional tree, and behavior as state-transition graphs.

The function and data class taxonomies are used as background knowledge, making possible the indexing of functions, or to compute the distance between data objects based on their data class. These taxonomies must be built by the knowledge engineer.

The knowledge base also stores a set of adaptation plans, which are records of previous adaptation operators used to solve functional differences by the system reasoning mechanism. It can be seen as a case base of adaptation episodes, which will enable faster adaptation operations in cases to solve new problems.

The CBR module comprises seven different parts, corresponding to the several steps in CBR: problem definition, problem elaboration, case retrieval, case adaptation, solution verification, solution evaluation, and case learning.

In the problem definition, the problem is specified in terms of functions, parameters, and other requirements. The problem definition is done at an abstract level, with the functional classification and parameter definition as major aspects of the problem definition.

The problem elaboration step is used when the functions are incompletely specified in terms of parameters. In this case, the system suggests several alternative sets of parameters that can be used as parameters of the function. This step can be seen as the design of the problem where the user can browse through similar problems selecting parameters from other functions to complete the problem specification.

Case retrieval is a crucial module of a CBR system, because if the "right" cases are not retrieved from the case base, the system will not be able to solve the problem. "Right" case can mean most similar case, more reusable case, or another definition. It is important that the system have the right definition. In our system, we make use of an

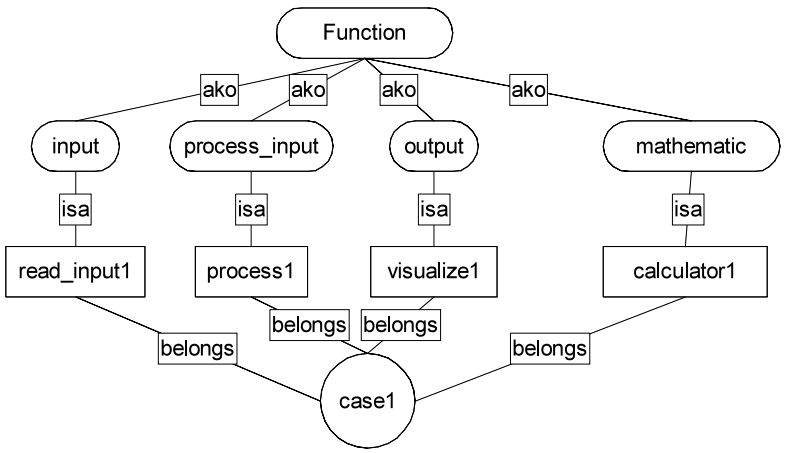


Fig. 2. An index structure example. The is-part-of links are not represented.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

indexing scheme to rapidly select the cases within the same functional classes or the closest cases regarding the function classes. After this first step, cases are then ranked, and for this we use a similarity metric, which will be described later.

The next step is to identify the functional differences between the retrieved case (or cases, if the user wants several alternative solutions) and the problem description. Once this is done, the system uses a blackboard algorithm (Engelmore & Morgan, 1988) to try to remove or solve the differences. The blackboard algorithm uses adaptation operators as knowledge sources, opportunistically applying the adaptation operators to the retrieved case. The result is a new case that solves the specified problem, and an adaptation plan that can be later used in a similar problem.

After the creation of the new case, it has to pass by a verification step that will check for inconsistencies. If failures are detected in the new case, then the verification module will try to repair the new case; otherwise the system can go back to the case retrieval phase selecting another case.

In the evaluation phase the new case is evaluated based on its performance, comparing it to other cases or using performance metrics.

Finally the case learning module stores the new case if it verifies learning criteria, which is only to store cases that are considered to be different from the ones in the case library (two equal cases in the case library will only degrade the system performance).

4. FUNCTION-BEHAVIOR CASE REPRESENTATION

The FBCR is used for representation of software programs. This formalism is intended for procedural software languages, like C, or more specific languages like VHDL. FBCR is derived from the structure-behavior-function (SBF) models developed by Goel (1991, 1992). SBF models were developed for representation of devices in which substances flow through physical structures. Within our formalism, substances are data objects and structures are the language operators and instructions.

Because software programs can be viewed as designs, the FBCR describes a software program along two abstract levels. One is the functional level of the software program, which specifies the purposes of the design. Another is the behavior level, and describes how the design functions are achieved by the structure. The structure is represented by the software program itself. In the remainder of this section, we describe how function and behavior are represented in the FBCR formalism.

4.1. Function

Within the FBCR formalism, the functional description of a design is a tree of functions. Each node of the tree represents a function. Functions are related to each other by partonomic links. This allows a function to be decomposed into subfunctions, providing a functional decomposition view of the software design.

The function tree represents two types of functions depending on whether they are leaves of the functional tree or at the other levels of the tree. Functions in a higher level are defined by the set of subfunctions and possibly by behavior graphs. Functions at the leaves of the tree are defined by behavior graphs. The function representation allows the software designs to be described at different levels of abstraction, from a conceptual level down to the instruction level.

A function is described by an identifier name, input data, output data, behavior, class, auxiliary data, and subfunctions. Input, output, and auxiliary data are sets of data objects. Data objects represent memory locations, and are variables, constants, and parameters in the program. A data object is defined by an identifier, a data class (for example the variable data type) and a set of properties. A property is described by the property name, value, and measure units. Input and output data objects represent, respectively, the function input and output parameters. Usually, associated with an output data parameter there is an output expression that corresponds to a possible final expression value of the parameter. This output expression is used for computing the similarity of parameters, and, indirectly, the functionality of a case. But it is also used for adaptation proposes, guiding the adaptation. Not all output parameters have output expressions, depending mainly on the abstraction level of the function to which the parameter belongs. Auxiliary data represent variables and constants, local to the function. Associated with each function is a class that belongs to a function taxonomy built to categorize functions and help the designer specify the problem. This taxonomy is created by the knowledge engineer.

4.2. Behavior

The behavior of a function is described by a graph comprising nodes and edges. Each node represents a behavior state, and an edge represents a transition between behavior states. A behavior graph represents the program execution flow, from the initial behavior state to the final behavior states, along which data objects are transformed. Data object properties can change, or data objects can be created or eliminated. A behavior graph is identified by an identifier and its initial state.

A behavior state represents the system state at a specific instant of time and is defined by an identifier, data objects, and the transitions that have this state as the source state.

Behavior transitions represent the causes and constraints for state changing. An identifier, a source state, a destination state, a set of causes, and a set of constraints define each behavior transition. In the next subsection, we explain the causes and constraints associated with behavior transitions.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

In Figure 3, we show the behavior graph for a function that computes the factorial of a number. Causes of transitions are operators, test instructions (like *if* statements), assignment instructions, or cycle instructions (like *for* statements), and are named primitive functions. These are the behavior building blocks in the graph, and each primitive instruction corresponds to a specific subgraph pattern, for example, *if* statements generate a bifurcation in the behavior graph. Transition constraints are represented as Boolean expressions. These constraints correspond to the data and property constraints described in the next subsection. Cycles are represented as loops in the behavior graph, for instance, the one comprising *State 6* through *9*. Some primitive functions such as "=" are responsible for changes in data object properties (e.g., *Transition 1*). Others create alternative behavior paths (such as *if* or *for* primitive functions).

4.3. Transition causes and constraints

Behavior transitions have two types of attributes: constraints and causes. In the first case, constraints represent Boolean expressions that must evaluate to true for the transition to occur. There are two types of constraints in this category: data and property constraints. A data constraint is a Boolean expression defined by a data object, a relational operator, and a value. This type of constraint states that the data object value must comply with the constraint defined by the relational operator and value for the transition to occur. The relational operator and the value are optional; if they are absent, the constraint means that the data object must exist. Property constraints are defined by a data object, a property, a relational operator, and a value. This type of constraint states that the data object must comply with a specific property value. In the case that the relational operator and value are omitted, the constraint implies the existence of the data object property. As an example, in Figure 3, a data constraint appears in *Transition 2*, meaning that this transition only occurs if the value of data object *n* is greater than zero.

Causes of transitions are named primitive functions. These are low-level functions representing language instructions, operators, or predefined functions. These constraints are specific to the software language in which the program is written. The primitive functions play an important role in the FBCR formalism, connecting the behavior level to the structural level. They make possible the conversion of behavior graphs into software programs and vice versa. In Figure 3, primitive functions are *if* statements, *for* statements, assignment statements, and operators $*$ and $++$.

4.4. Structure

The design product within our framework is the program written in the chosen software language. In the FBCR formalism associated with each case is the program code file, representing the design structure. The FBCR formalism has been developed to represent procedural languages, but we intend to extend it to object-oriented languages.

5. SIMILARITY AND REUSABILITY METRICS

In CREATOR II the user specifies a set of functional specifications (new problem) that are used as probes to search the case library of previous software designs. Then the sys-

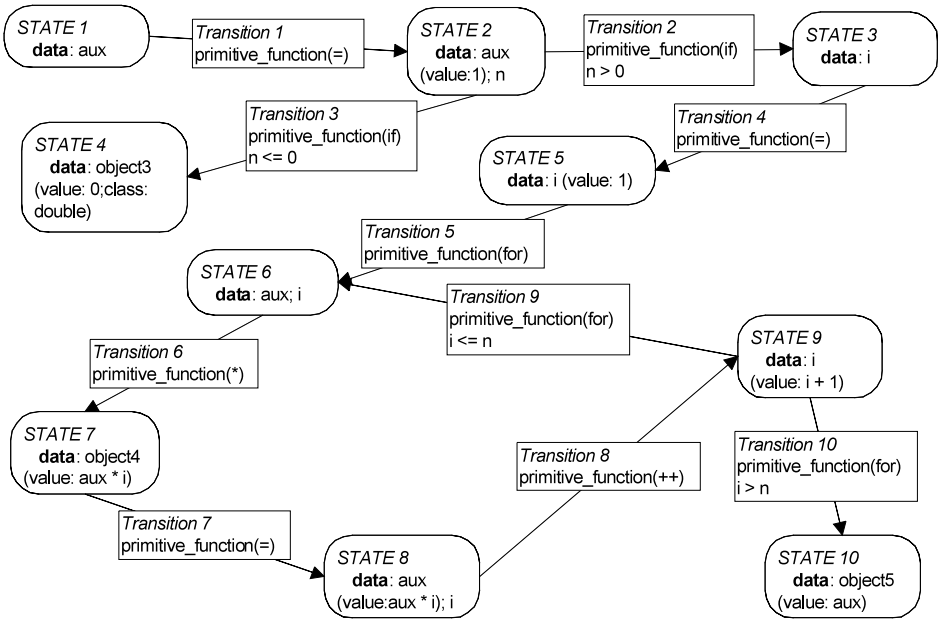


Fig. 3. The behavior graph of function *factorial*.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

tem retrieves a set of cases ordered by the similarity with the new problem. To rank the retrieved cases, CREATOR II uses a similarity metric function. As part of this metric, there is a reusability measure that estimates the amount of effort required for case reuse. In the remainder of this section we will describe how to compute the case similarity, the function similarity, and the reusability metric. We will also present some examples of its use.

5.1. Case similarity metric

The case similarity metric has three aspects: structural similarity, subfunction similarity, and functional similarity. To access the similarity of two cases, a mapping between their functional structure must be done. The mapping is performed by levels, starting in the higher-level functions of the problem description, recursively going down in the functional structure (which can comprise several tree-like structures). The mapping establishes for each function in the problem description (at the same tree level) a matching function in the retrieved case. There can be unmatched functions in the problem or in the retrieved case. The mapping algorithm is based on the distance between the functions classes, mapping the functions which have similar function classes.

The algorithm for computing the case similarity metric is shown in Figure 4.

In the algorithm, *case_ functions* and *problem_ functions* are, respectively, the retrieved case functions and the new problem functions. The first time that this algorithm is called these two variables must have the high-level functions of the case and the problem. The first step of the algorithm is to map the functions, after that the *case_sim* variable is initialized to zero. This variable is going to store the case similarity result. Then a cycle is used to compute the function similarity (function SIM, which will be described in the next subsection) between each mapped function and its subfunction similarity (the recursive call of CASE_SIM, with the subfunctions of each mapped function as parameters). This recursive call enables the computation of the structural similarity between cases. The final step in computing the case similarity is using the number of unmatched functions (see last line in the algorithm). α, δ, and β are constants defined by the user.

5.2. Function similarity metric

The function similarity is computed based on the function class distance, parameter similarity, and code complexity (or reusability metric). Functions in the same class are supposed to have the same functionality. The parameter similarity is computed based on the differences between the function parameters. Code complexity is a measure of function reusability (in the next subsection we will describe this metric in detail). The function similarity between *F1* (case function) and *F2* (problem function) is given by

$$\begin{aligned}\mathrm{Sim}(F1,F2) = {} & W1 * [1 - \mathrm{ClassDistance}(C1,C2)] \\ & + W2 * \mathrm{SimParams}(F1,F2) + W3 \\ & * \mathrm{Reusability}(F1),\end{aligned} \tag{1}$$

where *W1*, *W2*, and *W3* are constants defined by the user, and *C1* and *C2* are the function classes of *F1* and *F2*.

The function class similarity (ClassDistance) is the minimum path distance between the classes. A path distance between two classes has a length equal to the sum of the arc weights between these two classes. Weights for an arc are computed using a search algorithm that, when it must ascend in the class taxonomy, gives a weight *w1* to the arc, and when descending, gives the arc a *w2* weight. The weights used (*w1* and *w2*) are established by the user. What is computed by this distance is the number of specializations and generalizations needed to reach one function class, starting at another class. The formula for computing the similarity between function classes *C1* and *C2* is,

$$\mathrm{ClassDistance}(C1,C2) = 1 - [1/(\log(\mathrm{MinPath}(F1,F2)+1)+1)]. \tag{2}$$

Parameter similarity is based on the classification of the parameters within three different classes:

```
CASE_SIM (case_functions, problem_functions)
MAP_FUNCTIONS(case_functions,problem_functions)
case_sim  0
For Each Mapped Pair of Functions (function1, function2)
        case_sim  case_sim + α* SIM(function1,function2)
        sub_functions1  GET_SUBFUNCTIONS(function1)
        sub_functions2  GET_SUBFUNCTIONS(function2)
        case_sim  case_sim + δ* CASE_SIM (sub_functions1,sub_functions2)
End For
case_simcase_sim+β*(#unmatched_functions)/(MAX(#case_functions,#problem_functions))
```

Fig. 4. Algorithm for computing the case similarity metric.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

- Input control, if the parameter is used in a constraint of a transition of the function behavior graph, in other words, makes part of a condition that influences the program control flow; or if it is used to compute the value of a variable present in a constraint of a behavior transition;
- Input data, if it is used to compute the value of an output parameter, directly or indirectly;
- Output data, if it is an output parameter whose value is changed within the function.

Figure 5 shows an example of the parameter classification.

The classification of function parameters is important for the similarity metric, because it introduces the behavior aspect into account. When the user classifies the parameters associated with the desired functions, s/he is also stating what role the parameter is going to perform in the function.

To compute parameter similarity between two functions, *F1* and *F2*, we use the following function:

$$\begin{aligned}\text{SimParams}(F1,F2) = {} & Z1 * \text{SimPar}(F1_IC, F2_IC) + Z2 \\ & * \text{SimPar}(F1_ID, F2_ID) + Z3 \\ & * \text{SimPar}(F1_OD, F2_OD). \end{aligned} \quad (3)$$

FI_IC, *F2_IC* are the set of input control parameters in *F1* and *F2*, respectively, *FI_ID*, *F2_ID* input data, and *FI_OD*, *F2_OD* output data. *Z1*, *Z2*, and *Z3* are weights associated with each class of parameters, and can be defined by the user.

SimPar is the function similarity between two sets of parameters of the same type, and is given by the following formula:

$$\begin{aligned}\text{SimPar}(X,Y) = {} & T1 * (\text{MappingPar}(X,Y)/Mapped) + T2 \\ & * (UnMapped/(Mapped + UnMapped)). \end{aligned} \quad (4)$$

This function is computed using the mapping between parameters. *T1* and *T2* are weights associated with the mapped parameters and unmapped parameters respectively.

```
int f1(int a,int b,int c){
int aux;
if(c>0){
        aux=b/2;
        if(aux>7)
                return a;
        else
                return aux;
}
}
```

Fig. 5. A function example in C. Note that parameter *a* is input data, *b* is input data and input control, *c* is input control, and the return parameter of the function is data output.

Mapped and *UnMapped* are respectively, the number of mapped parameters and unmapped parameters.

The mapping is performed by the MappingPar function. This function tries to map a parameter in the retrieved functions set (X) to a parameter of the problem functions set (Y). This mapping is done for the parameter's data class and its data length. The data length refers to data objects that are arrays or vectors and have a data length associated with them. According to these two attributes, the mapping function tries to match the parameters with the same data class and data length, then it tries to match the remaining parameters with the same data class, and finally it tries to match the remaining parameters using the distance between two data classes. A parameter can only be mapped once, so if the sets have a different number of parameters, there will be unmapped parameters. The mapping function is given by

$$\text{MappingPar}(X,Y) = \text{Sum of all Map}(x,y), \quad (5)$$

where the $\text{Map}(x,y)$ is the mapping function between parameter x and parameter y. This function is given by

$$\begin{aligned}\text{Map}(x,y) = {} & CS1 * (1 - \text{ClassDistance}(x,y)) \\ & + CS2 * ((1 - |Lx - Ly|)/\text{Max}(Lx,Ly)), \end{aligned} \quad (6)$$

where *CS1* and *CS2* are weights concerning the distance between data classes, and the difference between data lengths; $\text{ClassDistance}(x,y)$ is the distance in the data class taxonomy between data class of x and data class of y; and *Lx* and *Ly* are the data length of x and y, respectively.

If the type of parameters being mapped is output data, then the comparison between the output expressions of both parameters is made. This is done by adding a new term to formula (6), which is

$$CS3 * [1 - \text{OutputExpDistance}(x,y)],$$

where $\text{OutputExpDistance}(x,y)$ is the distance between the output expressions of both parameters, and *CS3* is the weight associated with the output expression similarities.

5.3. Function reusability measure

The reusability metric is used in the function similarity metric to estimate the function reusability. This measure relies on three different function characteristics: the number of code lines of the function implementation, the number of comment lines in the function file, and the measure of the complexity of the function's behavior graph. Suppose that the reusability of *F1* is to be computed; then

$$\begin{aligned}\text{Reusability}(F1) = {} & K0/(K1 * Code + K2 \\ & * Behavior_Complexity + K3 * Comments), \end{aligned} \quad (7)$$

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

where *Code* and *Comments* are, respectively, the number of lines in the function code and the number of comment lines in the code. We have chosen the number of comment lines in the function file because much of the code files that were analyzed use global comments concerning all the functions in the file, and not just local comments to the function. Besides this, comments related to a specific function are very hard to identify inside a file; they can be anywhere. *K0*, *K1*, *K2*, and *K3* are user defined parameters that concern each part of the formula. This way the user can give more or less importance to a component of her/his choice.

Behavior_Complexity estimates the complexity of the function. This measure is given by the sum of all the function behavior graphs, plus the sum of each graph complexity for each subfunction. This is used in the function similarity metric. The complexity of a behavior graph is given by the following formula:

$$\text{Graph_Complexity} = C1 * \text{Nodes} + C2 * \text{Edges} + C3 * \text{Ramifications} + C4 * \text{Cycle_Nodes}, \quad (8)$$

where *Nodes* is the number of behavior states in the graph; *Edges* is the number of behavior transitions in the graph; *Ramifications* is the number of behavior states that have more than one transition starting from it; *Cycle_Nodes* is the number of behavior states that make part of a cycle; and *C1*, *C2*, *C3*, and *C4* are weights associated with each attribute, and are also user defined.

5.4. Example of the case similarity metric

We now present an example of how to compute the case similarity metric between a case C1 and a new problem P1. As seen in Figure 6, P1 comprises four functions, which are organized in a tree-like structure where F1 is the superfunction of F2, F3, and F4. In the figure we can see also the function class of each function, for example, F1 belongs to the functional class C1. The functional description of C1 is similar in structure to P1, except for the subfunctions of F14.

The algorithm for computing the case similarity starts with the high-level functions of both the case and the problem: Case_Sim({F11},{F1}). For CREATOR II to compute the associated value, it must first do the mapping between the two sets of functions, which in this case is trivial; F11 maps directly to F1. Then the functional similarity between F11 and F1 must be computed along with the subfunction similarity. This gives

$$\begin{aligned}&\text{Case_Sim}(\{\text{F11}\},\{\text{F1}\})\\ &\quad = \alpha * \text{Sim}(\text{F11},\text{F1}) + \delta\\ &\qquad * \text{Case_Sim}(\{\text{F12},\text{F13},\text{F14}\},\{\text{F2},\text{F3},\text{F4}\}) + \beta * 0.\end{aligned}$$

Since there are no unmatched parameters, the third term is zero. We now have to compute Case_Sim({F12,F13,F14}, {F2,F3,F4}), and for this we have to map the case functions into the problem functions. This mapping gives: F12 maps to F2, F13 maps to F3, and F14 maps to F4. From this,

$$\begin{aligned}&\text{Case_Sim}(\{\text{F12},\text{F13},\text{F14}\},\{\text{F2},\text{F3},\text{F4}\})\\ &\quad = \alpha * [\text{Sim}(\text{F12},\text{F2}) + \text{Sim}(\text{F13},\text{F3}) + \text{Sim}(\text{F14},\text{F4})]\\ &\qquad + \delta * \text{Case_Sim}(\{\text{F15},\text{F16}\},\{\quad\}) + \beta * 0,\end{aligned}$$

where Case_Sim({F15,F16},{ }) $= \alpha * 0 + \delta * 0 + \beta * [2/\text{Max}(2,0)] = \beta$. Then the case similarity between C1 and P1 is

$$\begin{aligned}&\text{Case_Sim}(\{\text{F11}\},\{\text{F1}\})\\ &\quad = \alpha * \text{Sim}(\text{F11},\text{F1}) + \delta * [\alpha * [\text{Sim}(\text{F12},\text{F2})\\ &\qquad + \text{Sim}(\text{F13},\text{F3}) + \text{Sim}(\text{F14},\text{F4})] + \delta * \beta].\end{aligned}$$

As can be seen by the final value and having in mind that the constants that appear in the formula are real values between zero and one, we can see that the function similarity of the high-level functions has more relevancy than other functions.

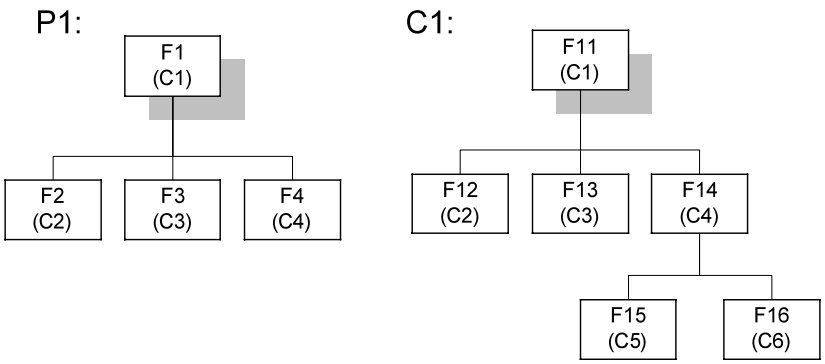


Fig. 6. The functional description of the new problem P1 and the case C1.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

5.5. Example of the function similarity metric

In this subsection we present an example of how to compute the similarity value between functions within the same class. Consider the two C functions in Figure 7. These functions compute the sum of the elements of a list ($F1$) and of an array ($F2$). They are considered to be in the same function class. Imagine that a new function to sum the elements in two lists is required. This new function is called *F3* and it is used for comparison with *F1* and *F2*, deciding which one is more similar to the new required function.

In Figure 8, there is the classification of the parameters in functions *F1*, *F2*, and *F3*. For example *F1* has *p* as input control parameter, and *p* belongs to the int^* data class, and is considered to have a data length of 1. Notice that when the parameter is a list, it is classified also as control input because it is used for the control of the *while* loop.

To rank functions *F1* and *F2* in relation to the similarity they have with *F3*, we need to compute $\mathrm{Sim}(F1,F3)$ and $\mathrm{Sim}(F2,F3)$, which can be calculated in the following way:

$$\begin{aligned}\mathrm{Sim}(F1,F3) &= W1 * [1 - \mathrm{ClassDistance}(F1,F3)] + W2 \\ &\quad * \mathrm{SimParams}(F1,F3) + W3 * \mathrm{Reusability}(F1) \\ &= W1 + W2 * 0.825 + W3 * 0.482 = 0.5 + 0.3 \\ &\quad * 0.835 + 0.2 * 0.482 = 0.8469.\end{aligned}$$

We consider the $\mathrm{ClassDistance}(F1,F3)$ to be equal to 0, or, that the function classes of *F1* and *F3* are the same. With *W1*, *W2*, and *W3* equal to 0.5, 0.3, and 0.2,

$$\begin{aligned}\mathrm{SimParams}(F1,F3) &= Z1 * \mathrm{SimPar}(\{p\},\{p1,p2\}) + Z2 \\ &\quad * \mathrm{SimPar}(\{p\},\{p1,p2\}) + Z3 \\ &\quad * \mathrm{SimPar}(\{y\},\{y\}) \\ &= 0.4 * 0.75 + 0.3 * 0.75 + 0.3 * 1 = 0.825,\end{aligned}$$

since we consider $Z1 = 0.4$, $Z2 = 0.3$, and $Z3 = 0.3$.

Because the data output parameters have the same description, they map exactly which gives a similarity score of 1. The similarity between input data and input control parameters is the same and is computed in the following way:

```
int F1(int *p){
   int sum=0;
   while(p){
      sum+=p->value;
      p=p->next;
   }
   return sum;
}
```

```
int F2(int n,int *t){
   int i;
   int sum=0;
   for(i=0;i<n;i++)
      sum+=t[i];
   return sum;
}
```

Fig. 7. C functions *F1* and *F2*.

$$\begin{aligned}&\mathrm{SimPar}(\{p\},\{p1,p2\}) \\ &\quad = T1 * (\mathrm{MappingPar}(\{p\},\{p1,p2\})/Mapped) \\ &\qquad + T2 * (UnMapped/(Mapped + UnMapped)) \\ &\quad = 1 * (1/1) - 0.5 * (1/(1+1)) = 0.75.\end{aligned}$$

The mapping between *p* and *p1* is 1, because they have the same data class and data length. So *Mapped* is 1 and *UnMapped* is also 1, and considering $T1 = 1$ and $T2 = -0.5$. The reusability value for *F1* is given by

$$\begin{aligned}\mathrm{Reusability}(F1) &= (K1 * Code + K2 * Behavior_Complexity \\ &\quad + K3 * Comments)/ \\ &\quad (Code + Behavior_Complexity + Comments) \\ &= (0.6 * 8 + 0.3 * 5.2 - 0.2 * 0)/(8 + 5.2 + 0) \\ &= 0.481818.\end{aligned}$$

K1 is 0.6, *K2* is 0.3, and *K3* is -0.2. Function *F1* has eight code lines, zero comment lines, and the behavior complexity is computed based on the behavior graph, which is depicted in Figure 9a (note that the graph is in a simplified version; only relevant information is present in the figure). Knowing that the behavior graph of *F1* has seven nodes, seven edges, and one node with more than one transition departing from it, that four nodes belong to a cycle, and that *C1*, *C2*, *C3*, and *C4* are, respectively, 0.3, 0.3, 0.2, and 0.2, the graph complexity of *F1* is:

$$\begin{aligned}\mathrm{Graph_Complexity} &= C1 * Nodes + C2 * Edges + C3 \\ &\quad * Ramifications + C4 * Cycle_Nodes \\ &= 0.3 * 7 + 0.3 * 7 + 0.2 * 1 + 0.2 * 4 = 5.2.\end{aligned}$$

In the same way, to determine the similarity between *F2* and *F3* we have that

$$\begin{aligned}\mathrm{Sim}(F2,F3) &= W1 * [1 - \mathrm{ClassDistance}(F2,F3)] + W2 \\ &\quad * \mathrm{SimParams}(F2,F3) + W3 * \mathrm{Reusability}(F2) \\ &= 0.5 + 0.3 * 0.785 + 0.2 * 0.475 = 0{,}8305,\end{aligned}$$

$$\begin{aligned}\mathrm{SimParams}(F2,F3) &= Z1 * \mathrm{SimPar}(\{n\},\{p1,p2\}) + Z2 \\ &\quad * \mathrm{SimPar}(\{t\},\{p1,p2\}) + Z3 \\ &\quad * \mathrm{SimPar}(\{y\},\{y\}) \\ &= 0.4 * 0.65 + 0.3 * 0.75 + 0.3 * 1 = 0.785.\end{aligned}$$

The similarity between $\{t\}$ and $\{p1,p2\}$ is the same as between $\{p\}$ and $\{p1,p2\}$, because *p* and *t* have the same data class and data length. Between output data is the same as above. In the case of $\{n\}$ and $\{p1,p2\}$, we have

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

F1	F2	F3
Input Control: p(int*,1)	*Input Control*: n(int,1)	*Input Control*: p1(int*,1); p2(int*,1)
Input Data: p(int*,1)	*Input Data*: t(int*,1)	*Input Data*: p1(int*,1); p2(int*,1)
Output Data: y(int,1)	*Output Data*: y(int,1)	*Output Data*: y(int,1)

Fig. 8. The parameter classification of function *F1*, *F2*, and *F3*.

$$\begin{aligned}\text{SimPar}(\{n\},\{p1,p2\}) &= T1 * (\text{MappingPar}(\{n\},\{p1,p2\})/Mapped) \\ &\quad + T2 * (UnMapped/(Mapped + UnMapped)) \\ &= 1 * (0.9/1) - 0.5 * (1/(1+1)) = 0.65.\end{aligned}$$

The mapping between n and p is given by

$$\begin{aligned}\text{MappingPar}(\{n\},\{p1,p2\}) &= CS1 * (1 - \text{Class_Distance}(int, int*)) \\ &\quad + CS2 * (1 - (|1-1|/\text{Max}(1,1))) \\ &= 0.5 * (1 - 0.2) + 0.5 * (1 - 0) = 0.9.\end{aligned}$$

The reusability measure for *F2* in relation to *F3* is computed in the same way and is 0.475 and the graph complexity is equal to 5.

As can be seen, *F2* (0.475) has been estimated to be more easy to reuse than *F1* (0.481). Reusability(*F2*), which measures the complexity of function *F2* regardless of function *F3*, is smaller than Reusability(*F1*). This means that in general, *F2* is easier to reuse than *F1*. But, from the similarity metric we see that *F1* is more similar to *F3* than *F2*. This makes sense, since *F1* sums the elements of a list, while *F2* sums the elements of an array. Nevertheless, these two functions are very similar in the function they perform, which can be seen by the values given by the metric.

6. EXPERIMENTAL WORK

We now present some experimental work done with the CREATOR II retrieval module. To study the parameter influence in the functional similarity metric we computed the retrieval error in relation to a user-made case ranking, since the user ranking is considered to be the most adequate. Our goal is to determine if and how the three main similarity parameters (class similarity, parameter similarity, and reusability) influence the similarity metric. To study this influence we explored the relationship between the values given to the weights of the similarity metric (*W1*, *W2*, and *W3*). Without appropriate values for these weights the similarity metric is useless. Because the similarity metric is integrated into a software design aiding tool (CREATOR II), the best referential to evaluate the similarity metric performance is the user rankings. We used a case library with 98 VHDL cases. These cases describe mainly small scale integration circuits (SSI). For an example of a VHDL file see Figure 10.

The problem given to the system is a Nand gate with two inputs and one output. The case library comprises various logical gates, and from these, 16 are retrieved by CREATOR II and then ranked by the case similarity metric and by the user. The user was presented with the problem and after browsing the case base, retrieved a set of cases and ranked them. We used 10 different parameter configurations for the function similarity metric, and then computed the error between the case similarity ranking and the user ranking. For this we used the following method: given

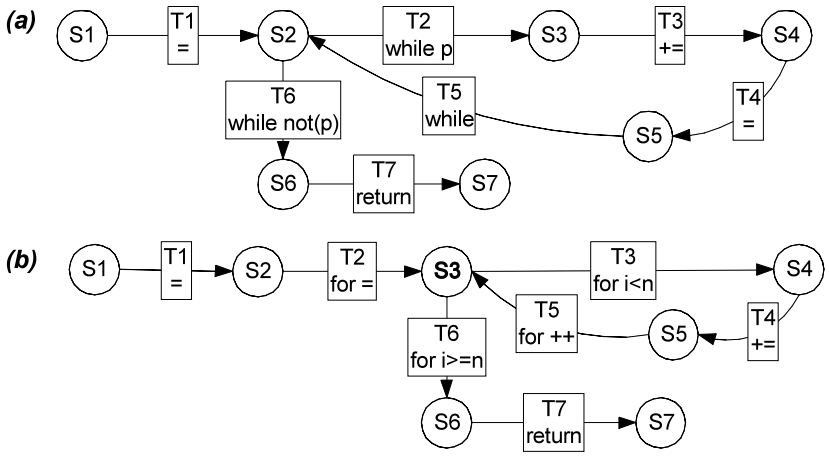


Fig. 9. The simplified behavior graphs of function *F1* (a) and *F2* (b).

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

```
----- bufif1_0 -----
-- 3 state bufif1 with no delay --
-----------------------------------
LIBRARY IEEE ;

USE IEEE.STD_LOGIC_1164.ALL ;
USE IEEE.STD_LOGIC_1164_EXTENSIONS.ALL ;

ENTITY bufif1_0 IS
 PORT (
    o1   : OUT  STD_ULOGIC ;
    i1   : IN   STD_ULOGIC ;
    c1   : IN   STD_ULOGIC
    ) ;
END bufif1_0 ;

ARCHITECTURE behav1 OF bufif1_0 IS
BEGIN
 bufif : PROCESS(i1,c1)
 BEGIN
  IF ( c1 = '1' ) THEN
    o1 <= i1 ;
  ELSE
    o1 <= 'Z' ;
  END IF ;
 END PROCESS bufif ;
END behav1 ;
```

Fig. 10. A VHDL file from the used library.

a case C1, $S_{\text{user}}(C1)$ is the ranking given by the user to case C1, and $S_{\text{system}}(C1)$ is the ranking given by CREATOR II to case C1. The error of a configuration is given by: $\sum [n + 1 - S_{\text{user}}(Cn)] * |S_{\text{user}}(Cn) - S_{\text{system}}(Cn)|$, where n is the number of retrieved cases. The first term in the sum accounts for the relative position of the ranking being evaluated, and the second accounts for the difference between the user's ranking and the metric's ranking. For each configuration, the system retrieved 16 cases. The computed error associated with each configuration is presented in Table 1.

Table 1. *The computed error associated with each weight configuration*

Parameter Configuration	Class Similarity (*W1*)	Parameter Similarity (*W2*)	Reusability (*W3*)	Error
C1	0.33	0.33	0.34	520
C2	0.80	0.10	0.10	181
C3	0.10	0.80	0.10	529
C4	0.10	0.10	0.80	627
C5	0.20	0.40	0.40	549
C6	0.20	0.60	0.20	541
C7	0.40	0.40	0.20	471
C8	0.60	0.20	0.20	284
C9	0.40	0.20	0.40	489
C10	0.20	0.20	0.60	579

The error value corresponds to the difference between the user's and the metric's rankings. To give an idea of the degree of difference between the user ranking and the metric ranking, configuration C2 has 10 functions in the wrong position, but the first eleventh ones are among the first 11 functions in the user ranking, thus having only a permutation among them. In contrast, configuration C4, which has the worst score, has only three cases in the right positions (first, fourth, and sixth cases). The metric always selected the best case correctly.

Analyzing the table values in a different way, we can see that there is a strong correlation between the error and the class similarity weight. As we see by Figure 11, a decrease in the class similarity parameter corresponds to an increase in the value of the ranking error. This was an expected result, since one of the most important aspects of a case similarity is the case functionality.



Fig. 11. The correlation between the class similarity parameter and the ranking error.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

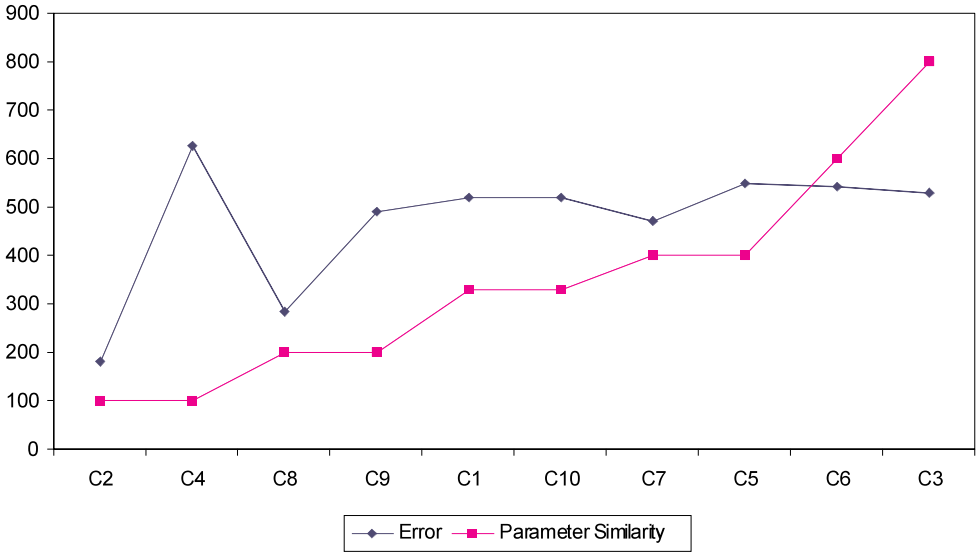


Fig. 12. The correlation between the parameter similarity and the ranking error.

We also did the same type of analysis for the parameter similarity and for the reusability. Figures 12 and 13 present the results obtained. The results are not conclusive, but they show a clear tendency for the error to stabilize in a high value with the increase of the respective weight value.

In a second set of experiments we fixed the function class similarity weight at 0.5 and assigned different values to the parameter similarity weight and to the reusability weight. Table 2 presents the values for the weights used in this experiment, and the respective ranking error. From Figure 14 we can infer that the error increases with the decrease of the reusability weight, thus increasing with the increase of the parameter similarity weight (for a *W1* value of 0.5).

Though our tests do not give us definite conclusions about the importance of each weight in the function similarity metric, they show us the main tendencies for these values and guide us in the assignment of values according to the user rankings. The experiments presented here were done with the same user; in future experiments we are planning to use different programmers.

7. RELATED WORK

The predecessor of CREATOR II was CREATOR, which was a case-based design system based on the SBF models. This system had a similarity metric built in that uses the

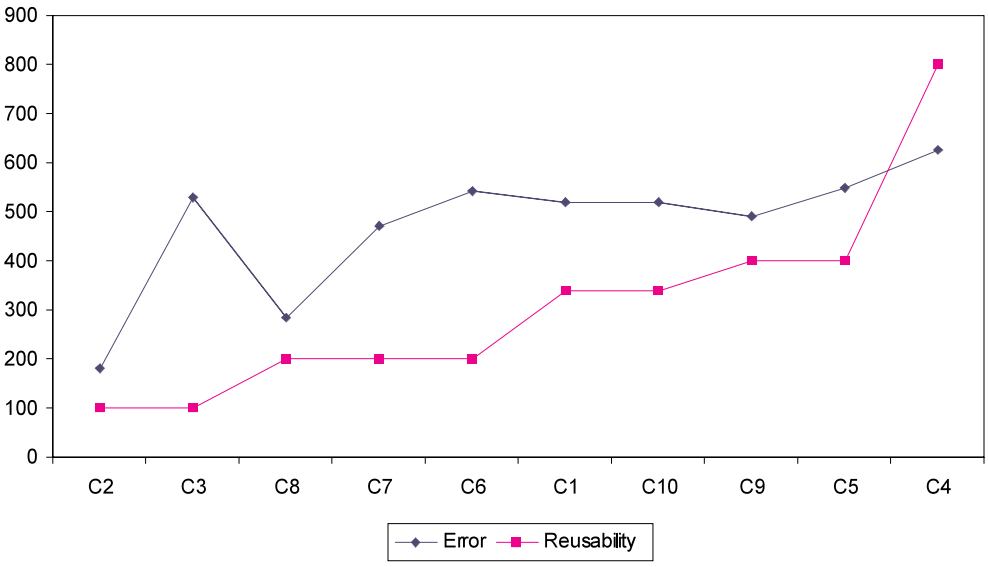


Fig. 13. The correlation between the reusability and the ranking error.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

Table 2. *The computed error associated with the parameter similarity and to the reusability in the second set of experiments*

Parameter Configuration	Parameter Similarity (*W2*)	Reusability (*W3*)	Error
C1a	0	0.5	546
C2a	0.1	0.4	391
C3a	0.2	0.3	277
C4a	0.3	0.2	226
C5a	0.4	0.1	235
C6a	0.5	0	132

similarities and differences between function input and output parameters (Gomes & Bento, 1997). This similarity is also used as a predictor for case adaptability, assuming that a function with more parameter differences is more difficult to adapt. Cases in CREATOR are represented by functions that are defined by the SBF formalism. The main difference between the similarity metric of CREATOR and CREATOR II is that the latter uses an improved reusability component in the similarity metric and it also adds new behavioral aspects to the similarity metric.

Kritik2 is a system developed by Ashok Goel that uses the SBF formalism to represent cases (Goel et al., 1997). In this system, after the retrieval of the relevant cases, they are ranked heuristically by ease of adaptation. These heuristics are a qualitative estimate of the effort needed to change the specifications of the retrieved case to deliver the desired functional specifications. This measure is based on the number of input and output behavioral states that match the new problem situation, and the degree of match in each behavioral state. In CREATOR II, the function similarity uses additional parameters to compute the reusability, such as code complexity and code comments. Besides this, CREATOR II uses a case similarity metric involving several functions, while Kritik2 only performs a function-to-function similarity.

These two ways of determining the function similarity are very similar, mainly differing in the way that this is done. While in CREATOR, this is done statically, using a network of cases, where each node is a case and an edge is a link representing the differences between the cases that it links, in Kritik2 this is done using a metric function.

Smyth and Keane (1995) developed the adaptation-guided retrieval (AGR) method, which ensures that adaptable cases are always retrieved. The goal of this technique is to minimize the amount of adaptation that is necessary in a typical problem-solving session. The AGR method involves three different steps. The first one is candidate selection, which is the stage where relevant cases are selected from the case base. The next step is the mapping of the target's features with those of the retrieved cases. In this step the local adaptability is assessed using a local adaptability metric. This metric is applied to each selected case, and it estimates the ease of adaptation in terms of applicable adaptation methods. These methods are predefined and have several heuristics associated with them. These heuristics are used to compute the adaptability metric for each retrieved case. The final stage is the assessment of global adaptability. The AGR method uses several high-level strategies to solve the interaction between local adaptation methods. To predict and estimate the amount of effort needed to solve situations of local adaptation-method interaction, the AGR uses a second metric that grades the retrieved cases according to the different repair methods that are suggested by each strategy.

The work of Smyth and Keane introduces new aspects to the methods of CREATOR and Kritik2. While CREATOR

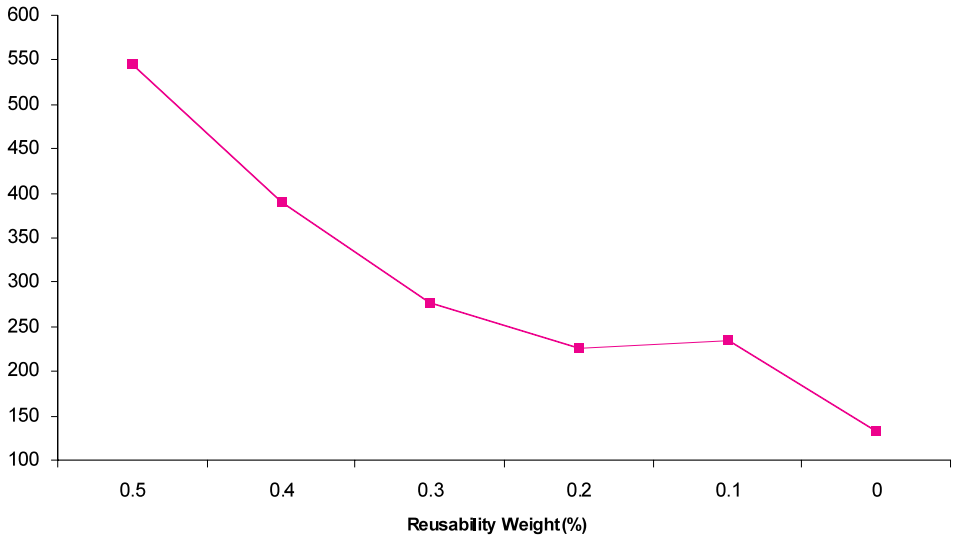


Fig. 14. The correlation between the reusability similarity parameter and the ranking error.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

and Kritik2 use only knowledge at the functional level to predict case similarity and to estimate case adaptability, the AGR method uses procedural knowledge, and the knowledge of what the adaptation methods can do, to estimate the adaptability of a case. All these works have in mind the CBR paradigm. Let us now look at a different work, developed in the software reuse domain.

Ruben Prieto-Diaz developed a framework for classification of software component reuse (Prieto-Diaz & Freeman, 1987; Prieto-Diaz, 1991). This framework is based in a faceted classification of software components. Selecting the most similar software component is done according to a classification scheme, where components are classified along six different facets. Ranking is then based on the ordering of the specifications by relevance. The component description format is based on a standard vocabulary of terms, imposes a citation order for the facets, and provides a metric for conceptual distances between terms in each facet. A conceptual graph is used to measure closeness among terms in a facet. Closeness is then measured by the closest path between two terms.

It then uses several metrics for estimation of the reusability of a software component. Each one of these metrics is related to one reuse attribute. Five reuse attributes were identified: program size, program structure, program documentation, programming language and reuser experience. For the program size, he used the number of code lines. The program structure is estimated based on the number of modules, number of links, and cyclomatic complexity. Program documentation is ranked along a subjective overall rating. Programming language is estimated based on the relative language closeness. The experience of the reuser is related to the proficiency level that she/he has in the programming language and the domain of application.

The work done by Prieto-Diaz uses new aspects of software reuse; it uses the structure aspect of software to estimate its reusability. None of the previous described works use this aspect; they all use only function to do this task. But the work done by Prieto-Diaz has a much simpler functional description, which does not allow the use of more complex and accurate reusability metrics at the functional level.

Within our work in CREATOR II, we join these two aspects: use of a functional description capable of deeper inferences on function similarity between cases, and use of structure attributes of the software components to estimate the ease of reusability. We also add a new aspect to the similarity metric between functions by using some behavioral knowledge to predict both function similarity and reusability. Classification of function parameters in input control, input data, and output data, is based on the function behavior.

8. CONCLUSIONS

Case similarity is a very important issue within CBR systems. But similarity should not be the only aspect to look at. When trying to solve a problem in the software reuse domain using a CBR approach, reusability of the selected software components should also be taken into account. In this article we propose a case similarity metric for software reuse and design based on the analysis of functional and behavioral knowledge, and also on the reusability of the software component. We also present an example illustrating the use of this metric, and how the choice of the associated weights must be done thoughtfully. We also present some experimental work with this metric showing the importance of function classification. We have also shown how previous works in this area relate to our own work.

Future work on CREATOR II includes developing a module for learning user preferences. This module is almost implemented and the goal is having the system learn the best weight configuration for the similarity metric according to user feedback. This is performed using an evolutionary strategy and is a way of fine-tuning the system to user preferences. This learning is performed off-line, when the system is not processing.

ACKNOWLEDGMENTS

This work was partially funded by the Portuguese Ministry of Science and Technology (MCT), under program PRAXIS XXI. We also thank the reviewers for their comments on the first version of this article.

REFERENCES

Engelmore, R. & Morgan, T. (1988). *Blackboard Systems*. Addison-Wesley.
Fouqué, G. & Matwin, S. (1993). Compositional software reuse with case-based reasoning. *Ninth Conf. on Artificial Intelligence for Applications (CAIA'93)*, IEEE Computer Society Press.
Goel, A. (1991). A model-based approach to case adaptation. *Thirteenth Annual Conf. on Cognitive Science Society (CogSci'91)*.
Goel, A. (1992). Representation of design functions in experience-based design. *Intelligent Computer Aided Design*, (Brown, D., Waldron, M., and Yosnikawa, H., Eds.). Elsevier Science.
Goel, A., Bhatta, S., & Stroulia, E. (1997). Kritik: An early case-based design system. *Issues and Applications of Case-Based Reasoning to Design*, (Maher, M.L. and Pu, P., Eds.). Lawrence Erlbaum Associates.
Gomes, P. & Bento, C. (1997). A retrieval method for exploration of a case memory. *Portuguese Conf. on Artificial Intelligence*, Springer-Verlag.
Kolodner, J. (1993). *Case-Based Reasoning*, pp. 37–48. Morgan Kaufman, Berlin.
Maher, M.L., Balachandran, M. & Zhang, D. (1995). *Case-Based Reasoning in Design*. Lawrence Erlbaum Associates.
Prieto-Diaz, R. (1991). Implementing Faceted Classification for Software Reuse. *Communications of the ACM*.
Prieto-Diaz, R. (1993). Status report: Software reusability. *IEEE Software* (May).
Prieto-Diaz, R. & Freeman, P. (1987). Classifying software for reusability. *IEEE Software*.
Rosenman, M. & Gero, J. (1998). Purpose and function in design. *Design Studies 19(2)*, 161–186.
Smyth, B. & Keane, M. (1995). Experiments on adaptation-guided retrieval in case-based reasoning. *Int. Conf. on Case-Based Reasoning (ICCBR'95)*, pp. 313–324. Springer-Verlag, Berlin.

Paulo Gomes got his B.S. and M.Sc. degrees in Informatics Engineering from the University of Coimbra, Portugal, and is a Ph.D. student in the University of Coimbra. He is

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press

currently Assistant Professor in the Informatics Engineering Department of the University of Coimbra. He is a member of the Coimbra Artificial Intelligence research group doing research in case-based design and software design and reuse.

Carlos Bento obtained a Ph.D. in Engineering Sciences, Specialty in Informatics, from the University of Coimbra. His thesis subject was reasoning based on cases imperfectly described and explained. He is currently Auxiliary Professor in the Informatics Engineering Department of the University of Coimbra. He is a member of the Coimbra Artificial Intelligence research group, where he is coordinator for the research areas of case-based reasoning and intelligent data analysis. He is also involved in the research area of creative systems.

https://doi.org/10.1017/S0890060401151061 Published online by Cambridge University Press