ELSEVIER

Science of Computer Programming 40 (2001) 213–234

Science of Computer Programming

www.elsevier.nl/locate/scico

Design-code traceability recovery: selecting the basic linkage properties

G. Antoniol, B. Caprile, A. Potrich, P. Tonella*

ITC-Irst, Centro per la Ricerca Scientifica e Tecnologica, I-38050 Povo (Trento), Italy

Accepted 2 February 2001

Abstract

Traceability ensures that software artifacts of subsequent phases of the development cycle are consistent. Few works have so far addressed the problem of automatically recovering traceability links between object-oriented (OO) design and code entities. Such a recovery process is required whenever there is no explicit support of traceability from the development process. The recovered information can drive the evolution of the available design so that it corresponds to the code, thus providing a still useful and updated high-level view of the system.

Automatic recovery of traceability links can be achieved by determining the similarity of paired elements from design and code. The choice of the properties involved in the similarity computation is crucial for the success of the recovery process. In fact, design and code objects are complex artifacts with several properties attached. The basic anchors of the recovered traceability links should be chosen as those properties (or property combinations) which are expected to be maintained during the transformation of design into code. This may depend on specific practices and/or the development environment, which should therefore be properly accounted for.

In this paper different categories of basic properties of design and code entities will be analyzed with respect to the contribution they give to traceability recovery. Several industrial software components will be employed as a benchmark on which the performances of the alternatives are measured. © 2001 Elsevier Science B.V. All rights reserved.

Keywords: Traceability; Object-oriented programming; Software evolution; Program understanding; Software maintenance

1. Introduction

The complexity of developing software systems is usually tackled by following a phased software development process, in which the activities performed within a phase refine the artifacts produced by the previous one. Requirement analysis, design and

* Corresponding author. Tel.: +39-461-314524; fax: +39-461-314591.
E-mail addresses: antoniol@ieee.org (G. Antoniol), caprile@itc.it (B. Caprile), potrich@itc.it (A. Potrich), tonella@itc.it (P. Tonella).

coding are typically present in almost any software development process, but a phased process does not automatically help to trace how requirements evolve into design and design into code.

The design is an important source of information about the code, especially when the system enters the maintenance phase [3]. However, maintaining consistency between design and code is a costly and tedious activity frequently sacrificed during development and maintenance due to market pressure. Changes made directly in the code are not necessarily reflected into the design, which therefore becomes obsolete very quickly.

As long as no critical design constraint is violated, design-code realignment is achieved, in this scenario, by producing an updated version of the design. In fact, it is the code that now contains the information necessary for the upgrade. Even if reverse-engineering tools can extract design representations from the code, it is preferable to evolve existing design so that it matches the code. In fact, designs produced by software engineers are usually richer than those extracted automatically, since they include context and high-level semantic information. Furthermore, in reverse-engineered designs, multiple semantics are candidate explanations for the same piece of code (e.g., in C++ a pointer may be used to implement an association as well as an aggregation). That is why evolved designs are considered of higher quality, provided that traceability with code is maintained.

The activity of checking the compliance and evolving the design can be greatly assisted by automatic tools. Few approaches and systems to monitor the implementation's faithfulness to its design have been proposed in the literature [13–15,19]. In [13], a language for annotating Ada programs is defined. Meyers et al. [14] designed and implemented a language, CCEL, based on assertions to express constraints on the structure and style of object oriented-programs implemented in C++. Murphy et al. [15] developed an approach, called the *software reflexion models*, in which the user provides a high-level model of the system and a map stating how entities in the high-level model should be associated to those found in the source code. Sefika et al. [19] proposed a hybrid approach which, by integrating logic-based static and dynamic visualization, helps determine design-implementation congruence at various levels of abstraction, from coding guidelines to architectural models such as design patterns [10] and connectors [11], to design principles like low coupling and high cohesion.

An in-field study of traceability for a system at Ericsson is described in [12]. The reported experience suggests that by emphasizing traceability as a quality factor from the very beginning, the documentation will be clearer and more consistent. However tracing items with no tool support or in models partially inconsistent and underdocumented requires significant effort.

This paper compares different traceability recovery methods, anchored on different basic class properties or their combinations. It substantially extends and complements the work described in [1,2], which is focused on the traceability procedure itself. The process operates on design artifacts expressed with the object modeling technique (OMT) [17] notation and accepts C++ source code. Both design and code are represented using a custom OO design description language, the abstract object language (AOL). The process recovers an "as-is" design from the code in AOL, compares the recovered design with the actual AOL design, and helps the user to deal with inconsistencies by providing a similarity measure for the matched classes and pointing out the unmatched ones. This activity was partially funded by Sodalia SpA[1] under the DEMOS 2 project, aiming at estimating software size and complexity, and improving its quality.

Bunge's ontology [4,5] has been taken as the base conceptual framework, upon which a similarity criterion was defined, to be employed for a design-code traceability check. According to [4,5], an object is viewed as an individual which possesses properties. Comparing individuals for similarity translates into checking the similarity of the individuals' properties. When instantiated in the context of OO design to code traceability, individuals become classes, while there is no obvious selection of the properties to be considered. Many different class properties can be chosen to drive the traceability recovery process. The properties which provide useful information for the recovery of traceability links are those that are maintained almost unchanged during the refinement of the design into the code. The ones selected for examination in this paper include the class name, the names of fields and methods, and their combination, and the relations that the class holds with other classes. Other useful properties are based on the class *dictionary*, consisting of the English words, acronyms and contractions that are employed as building blocks when defining the identifiers used by the class. A similarity measure between class properties is proposed for each set of properties under examination. After determining the similarity between atomic properties, the maximum match algorithm [8] is applied to extract the best matching attribute pairs (i.e., the pairing of properties which maximizes the overall similarity measure). A summary similarity between the compared classes is then computed as the average similarity measure of their attributes. The desired traceability links between design and code are retrieved by applying the maximum match algorithm at the class level. A further step can be performed to separate matched classes from unmatched ones, by means of a maximum likelihood classifier [9].

Each selection of a set of class properties, to be used as traceability anchors, gives origin to a traceability recovery method. Such traceability recovery methods were used in experiments on industrial design and code. Support tools have been developed to extract the "as-is" design from source code, to match design into code and finally for result visualization. The focus of this paper is the comparison of the proposed methods, with reference to their ability to recover many underlying traceability links with few errors. The advantages and disadvantages of choosing a given set of properties, and of combining properties to produce new traceability recovery methods, are discussed by considering the performances of each method on the available data set. Moreover, the overlapping of the examined methods is analyzed to understand the extent to which they are independent and can be used in combination.

[1] Sodalia SpA is one of the leading telecommunication software companies in Italy.

The paper is organized as follows: in Section 2 the adopted traceability recovery procedure is summarized, and all the notions needed for a full understanding of the following sections are provided. Section 3 describes the different class properties, and their combinations, that can be employed to determine the basic traceability links in the recovery process. The associated methods are described and the rationale for choosing each one is given. Section 4 is devoted to the experimental results obtained on a set of industrial software components for which both design and code are available. The different traceability recovery methods proposed earlier are compared on an industrial benchmark, and their performances are evaluated in terms of the accuracy, meant both as the ability of committing no errors and of extracting all traceability links. Their relative coverage of the links to be reconstructed is also considered. Section 5 discusses the experimental results trying to gain general lessons that can be used to repeat the study in a different context. Moreover, some ideas on the transformation of the design artifacts implemented in the code are provided. In Section 6, our approach is compared with related work in design-code compliance verification. Finally, in Section 7 conclusions are drawn.

2. Traceability recovery

The traceability recovery procedure is described in detail in [1,2]. It is summarized below to provide the reader with all the information needed to fully understand the core parts of this paper.

2.1. Similarity between object properties

Design and code entities can be considered from an ontological perspective. According to Bunge's ontology, objects can be viewed as *substantial individuals* which possess *properties*. Chidamber and Kemerer [7] proposed to represent an object X as the following pair:

$$X = \langle x, P(x) \rangle,$$

where X is identified by its unique identifier x, and $P(x)$ is its finite collection of properties.

In general, two objects X and Y may possess different properties. Thus, a preliminary step in the definition of a similarity measure between them is the introduction of a map m between a subset of the properties of X and a subset of the properties of Y, to be considered as matched properties. Then the remaining properties from $P(x)$ and $P(y)$ are unmatched properties of X and Y, respectively:

$$\begin{aligned}
&m : P(x) \to P(y),\\
&Unmatched(X) = P(x) - Dom(m),\\
&Unmatched(Y) = P(y) - Ran(m),
\end{aligned}$$

where m is an injective function from $P(x)$ to $P(y)$, *Dom* is the domain and *Ran* the range. Unmatched properties of X are those not in the domain of m, while unmatched properties of Y are those not in the range of m.

Given a measure of similarity, $s(p,q)$, between the matched properties $p \in P(x)$ and $q \in P(y)$ of two different objects X and Y, an overall similarity measure between the objects can be obtained by applying a suitable average operator as, for example, the arithmetic average:

$$s(X,Y) = 1/|Dom(m)| \sum_{p \in Dom(m)} s(p, m(p)). \tag{1}$$

An overall picture of the similarity between X and Y is therefore given by the sets of unmatched properties (Unmatched(X), Unmatched(Y)) and by the average similarity measure between matched properties ($s(X,Y)$). More detailed information can be obtained by the individual similarity measures for the matched pairs of properties ($s(p,q)$, $p \in Dom(m), q = m(p)$).

2.2. Recovering design-code traceability links

When instantiating the above notion of similarity between objects to trace OO design into code, classes have to be considered as the basic entities. Several class properties can be considered as anchors for an evaluation of the similarity between two classes. Examples are the class name and the class attributes (fields plus methods). Some of the available alternatives are described in the next section.

Given a pair of classes for which a similarity measure has to be determined, the similarities of the contained properties are computed first. If, for example, the considered properties are character strings, a similarity measure can be derived by complementing a string edit distance. After computing the similarity between each pair of properties, the match function can be inferred by applying the maximum match algorithm [8] to the bipartite graph in which nodes are respectively properties from design and code, connected by edges that are weighted with the similarity measures. The edges computed by the algorithm as those giving the maximum match define the desired match function. A further outcome of this algorithm is the set of unmatched attributes in the design and code. Then an average similarity measure can be computed for the two classes, as the arithmetic average of the similarity measures in the edges selected by the maximum match algorithm.

By repeating the above procedure for each pair of classes, it is possible to determine their respective average similarity measure. To determine the correspondence between design and code at the class level, it is possible to exploit the maximum match algorithm again. In this case, the nodes in the bipartite graph are respectively classes from the design and code, while edges are weighted with the average similarity measures. The edges extracted by the algorithm represent the traceability links between design and code. Each link is weighted with a similarity measure. In addition, an initial set of unmatched classes is determined as those having no traceability link attached.

The presence of a traceability link between a class in the design and a class in the code is not sufficient to state that a match occurs. In fact, the similarity measure associated to the link may be very low, and the edge in the bipartite graph could have been selected by the maximum match algorithm only as an effect of maximizing the total match measure. Therefore, the links connecting matched classes from design and code have to be classified to distinguish truly matched classes from unmatched ones. The use of a maximum likelihood classifier [9] is proposed for such a purpose, since it returns an optimal threshold value for the similarity measure. Class pairs having similarity below this threshold value are to be considered unmatched, while those whose similarity falls above the threshold are matched classes.

The computation of the maximum likelihood threshold requires that a set of class pairs be labelled preliminarily as either matched or unmatched. For each of the two categories, the probability density has to be estimated from the empirical frequencies. The point where the two curves intersect is the optimal value of the threshold.

The accuracy of the classifier is then evaluated on a test set, different from the one used to compute the threshold. When few examples are available, the evaluation can be conducted with a *cross-validation* technique [20]. Each component in turn is considered as a test case, while the remaining components are used to determine the threshold. The test procedure is thus re-applied for each available component. Average performance and robustness can then be assessed on a wider base than a single test case.

3. Basic traceability anchors

Several class properties and property combinations can be chosen as basic anchors for the computation of a similarity measure. In Table 1 properties are listed that should be relatively stable when the design is refined into code. Therefore they are the best candidates to support traceability recovery. The labels for the corresponding traceability recovery methods are given in the left column of Table 1.

3.1. C, M, F, and A methods

The name of a class is the basic property to be considered when tracing the class from design to code. The names of methods and fields in a class can also provide useful anchor points to traceability. All of them are characters strings, so that a similarity measure between class properties can be defined with reference to the edit distance between strings described in [8]:

$$s(p_{\rm d}, p_{\rm c}) = 1 - d(p_{\rm d}, p_{\rm c})/(|p_{\rm d}| + |p_{\rm c}|), \tag{2}$$

where $p_{\rm d}$ and $p_{\rm c}$ are the strings associated to a property from the design and code, respectively, and d is the edit distance. The edit distance chosen for this work [8] counts the number of characters to be inserted or deleted in order to transform the first

Table 1
Traceability recovery methods based on different basic properties of the classes to be matched

Method	Class property
C	Class name
M	Class methods
F	Class fields
A	Class attributes (methods plus fields)
CA	Class attributes (methods plus fields) prefixed with class name, and augmented (if needed) with class constructor and destructor
D	Class dictionary
WD	Weighted class dictionary: similarity between classes is computed giving lower weights to the most frequent words
R	Class relations (inheritance and collaboration)
RC	Class relations (inheritance and collaboration) augmented with the name of the target class

string into the second one. Since the upper bound for such an edit distance is the sum of the string lengths ($|p_\mathrm{d}| + |p_\mathrm{c}|$), the above similarity measure is between 0 and 1, being 0 when the two strings p_d and p_c have no character in common, and 1 when they coincide.

Alternative edit distances could be adopted, but their selection requires an analysis of the building process for design and code class identifiers, which is out of the scope of this paper. The performances of the edit distance taken from [8] were judged satisfactory for the purposes of traceability recovery.

3.2. CA method

A combination of the traceability methods C and A is given by method CA, consisting of class attributes (methods and fields) names prefixed with the name of the class, and properly separated (e.g., the string `"A::f"` is associated to method `f` of class `A`). Since this combined property is still a string, the similarity measure based on the edit distance can be used in this case too.

To account also for those classes having no attribute specified in the design (attribute specification is optional), class constructors and destructors are added, if not already present, to the set of properties to be matched (e.g., `"A::A"` and `"A::∼A"`).

3.3. D method

The *dictionary* associated to a class is the set of English words, acronyms and contractions that are used by designers and programmers to build the identifiers used in the class. To determine the dictionary of a class, the identifiers in the class have to be segmented into the component words, that are then collected. The identifiers considered in this work for each class in the design and code are those defining the class itself and all its methods and fields. The segmentation procedure can be automated partially, by exploiting an English dictionary and collections of acronyms and contractions commonly adopted by software engineers. When the segmentation procedure cannot split the given string into words, a manual intervention is required to handle the string properly.

Dictionaries associated with design and code classes should be good anchors for the traceability links. The similarity measure on which their recovery is based is once more that in Eq. (2), since the considered atomic properties are still strings.

3.4. WD method

If the similarity measure associated with a pair of design and code classes is computed as the arithmetic average of the dictionary similarities (Eq. (1)), the same weight is given to frequent and infrequent words. The problem with this approach is that the presence of attributes common to many classes does not provide much information on the traceability between design and code. For example, the methods `set` and `get` are frequently included in a class. As a consequence, their presence in both the design and code should not be treated as the presence of an infrequent word, which would be a much stronger hint of a traceability link between the two classes.

Therefore an additional method based on the class dictionary is considered, that exploits a weighted average to compute the similarity between classes using the similarity between dictionary words:

$$s(X,Y) = \frac{1}{\sum_{p \in Dom(m)} w(p,m(p))} \sum_{p \in Dom(m)} w(p,m(p)) s(p,m(p)), \tag{3}$$

where $w(p,q) = 1/\bar{f}(p,q)$ and $\bar{f}(p,q) = (f(p) + f(q))/2$ (i.e., the weight for two words p and q from design and code classes is the inverse of their average relative frequency). The resulting overall similarity measure is still between 0 and 1, being 0 when all individual similarities are 0 and 1 when they are all maximum (1). In the intermediate cases, similarities of frequent words are considered to hold less information than infrequent ones, and the associated weight is consequently lower.

The structural relations between classes are another important traceability clue. Properties in methods R and RC deal with them.

3.5. R method

Each class may be connected to other classes through inheritance, association and aggregation relations. The inheritance relation is directed, so that it generates two properties, *generalizes* and *extends*, to be attached to the base and to the derived class, respectively. Association and aggregation are difficult to distinguish in the code, since they can be implemented with the same programming language constructs (e.g., pointers), even though they are usually represented with different formalisms in the design. For this reason, the two relations will be merged into a more generic *collaborates* property. Moreover, associations in the design are directed only as an option. Therefore they will be considered as attached to both the connected classes.

If the relations owned by a class are considered with no regard to the target class, they can be distinguished only by type, and different relations of the same type can only be enumerated. Aggregations and associations can be decorated optionally with a name in the design, but this is not mandatory and often such name cannot be actually found in the examples under analysis. For this reason, the match based on the relations exploits the relation type and not the name. Therefore a new similarity measure has to be defined for relation matches, given a pair of classes C_d and C_c from design and code:

$$s(C_\mathrm{d}, C_\mathrm{c}) = \frac{\min(NG_\mathrm{d}, NG_\mathrm{c}) + \min(NE_\mathrm{d}, NE_\mathrm{c}) + \min(NC_\mathrm{d}, NC_\mathrm{c})}{NG_\mathrm{d} + NE_\mathrm{d} + NC_\mathrm{d}}, \tag{4}$$

where $NG_\mathrm{d}, NE_\mathrm{d}, NC_\mathrm{d}$ and $NG_\mathrm{c}, NE_\mathrm{c}, NC_\mathrm{c}$ are the number of *generalizes*, *extends* and *collaborates* in the design and code classes, respectively (with N_d and N_c we will denote their number with no regard to the type). The rationale for Eq. (4) is that an increase in the number of relations is considered normal when transforming the design into the code, the latter being a refinement. Consequently, such an increase does not affect the similarity level ($\min(N_\mathrm{d}, N_\mathrm{c}) = N_\mathrm{d}$). On the contrary, when there are fewer relations in the code than in the design ($\min(N_\mathrm{d}, N_\mathrm{c}) = N_\mathrm{c}$), the resulting similarity is reduced. If none of the relations in the design can be found in the code ($N_\mathrm{c} = 0$), the associated similarity measure becomes 0.

3.6. RC method

More structural information on the class relations can be obtained if the target class of a relation is considered as a further specification of the relation. It is possible to represent these kinds of properties as a class attribute of type string. If, for example, class `A` generalizes class `B`, the attribute `"generalizes->B"` can be attached to class `A`, and the attribute `"extends->A"` can be attached to class `B`. The resulting attributes are a mix of relations, distinguished by type, and target class name. The string representation of such a mix allows the exploitation of the edit distance to compute the similarity measures, as done with the other string attributes of the classes.

The construction of a combined representation comprising both relation type and target class name should be handled with care. When the edit distance is computed to determine the similarity between two such attributes, the relative length of the first part of the string (representing the relation type) with respect to the second part of the string (representing the target class name) may affect the resulting similarity. If, for example, the string for the relation type is much longer than the class name, the relation type is consequently weighted much more than the target class name. In other words, relations of the same type with different targets produce high similarity values, while different relations within the same class produce low similarity values.

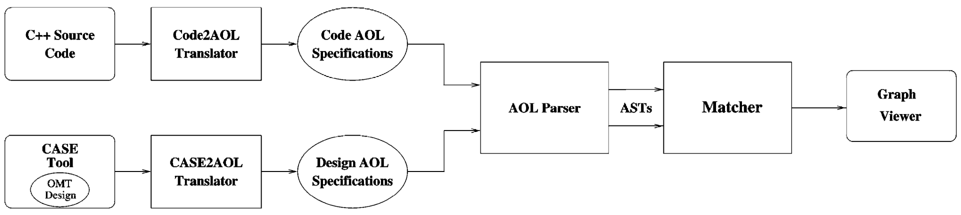

Fig. 1. Design-to-code matching process.

For the present work, the string for the relation type was chosen with a length approximately equal to the average length of the class names. Consequently relation type and target class contribute uniformly to the resulting similarity level.

4. Experimental results

The whole design-to-code traceability check process, represented in Fig. 1, consists of the following steps:

1. *AOL representation extraction*: the AOL textual representation can be recovered from both design and code through a `Case2AOL` and a `Code2AOL` translator, respectively.
2. *Representation parsing*: an `AOL Parser` produces the AST (Abstract Syntax Tree) on which subsequent phases rely.
3. *Match between design and code representations*: a `Matcher` module implements the traceability check; it includes a function to compute the similarity between class properties, an implementation of the maximum matching algorithm and a maximum likelihood classifier.
4. *Result visualization*: a `Pair Difference Coloring` module shows the results of matching graphically, highlighting the similarities and differences between classes in the design and code.

Finally, we come to the last step, in which the design is modified in order to solve all outlined differences from code. This phase cannot be completely automated since the reasons for the major differences have to be fully understood by the designer, in order to perform a meaningful update of the design.

Additional information can be obtained by querying the traceability links, and the associated measures, for the attributes of individual classes.

4.1. Experimental setup

To assess the approach in an industrial environment, design-code compliance check was conducted on the design and code of industrial software for telecommunications, provided by Sodalia SpA under the DEMOS 2 project. 29 components (about 308

KLOC [thousand lines of code]) were analyzed, for which both OO design models (stored as object models in the StP/OMT repository) and their corresponding code were available. All components were developed using the C++ language.

Given the threshold which allows classifying paired classes as matching or non-matching, two kinds of errors can occur. The first error is the classification of matching pairs as non-matching. The parameter which accounts for this error is the *recall*, computed as the ratio of truly matching class pairs classified as matching over the total number of truly matching class pairs. When recall is 1, no matching pair is missed by the classifier (no "missed alarms"). The second error is the classification of non-matching pairs as matching. The parameter accounting for it is the *precision*, computed as the ratio of truly matching class pairs classified as matching over the total number of pairs classified as matching. When precision is 1, no non-matching pair is classified as matching (no "false alarms").

The determination of the maximum likelihood classifier threshold requires that a set of components be extracted from the benchmark and used for that specific purpose. An evaluation of the methods can be performed only on the remaining components. Cross-validation was adopted to obtain the maximum information from the available test cases. Each component was in turn considered the test case, and the threshold was determined on the other ones, thus allowing a repetition of the evaluation over all available components. In this way the resulting number of method evaluations is not lower than the number of components, as occurs when the threshold is estimated once for all.

4.2. Performances of the methods

Fig. 2 shows the plots of the precision level produced by the different traceability recovery methods proposed in the previous section, based on the threshold computed by the maximum likelihood classifier. The name of each method is shown as a label at the bottom right of each plot, while the precision value is given over all the analyzed components, numbered on the horizontal axes. Note that the points on each graph are joined through straight lines solely to help the visual comparison of the curves.

The two curves with the highest precision levels are those for the C and CA methods. The respective shapes are very similar, thus indicating a similar ability of recovering traceability links over the analyzed components. The curve for the method M exhibits higher levels than the one for the method F, suggesting that method names are a better traceability indicator than the fields. When the two are combined (as in the A method), performances become worse than for the M method. The inaccuracies of the F method seem to affect the A method as well, rather than being compensated for by the M method.

When comparing curves for the D and WD methods, similar shapes can be observed, thus suggesting that weighting the words in the class dictionary does not produce a remarkable improvement of the performances. No precision data is available for the R method, and no associated curve can be plotted. The reason is that such a method

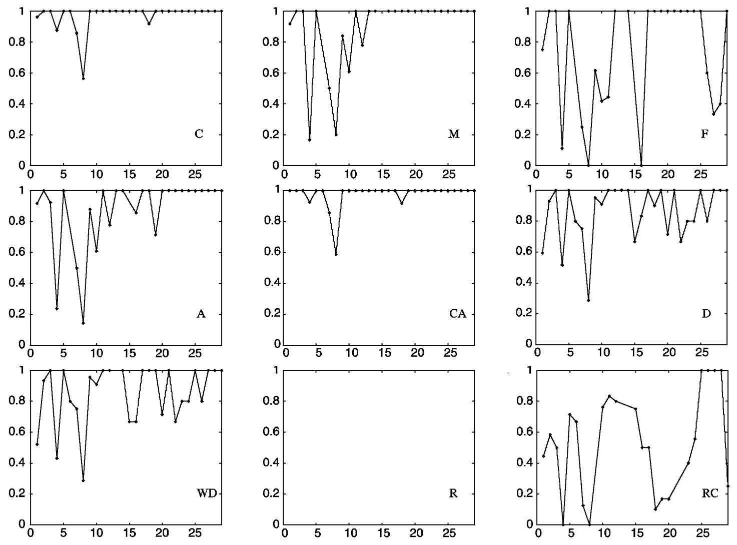

Fig. 2. Precision of different traceability recovery methods over all the analyzed components (on the horizontal axis).

generates two distributions of matched and unmatched class pairs over the similarity level both with a peak corresponding to 1. Consequently, the maximum likelihood classifier determines a value for the classification threshold equal to 1, which cannot classify any class pair as matched. In other words, even if the count of the relations in a class allows pairing matched classes correctly in some cases, the presence of many incorrect pairs with similarity 1 forces the classifier to increase the threshold. The final result is that no class pair is classified as matched since the two distributions cannot be separated by the high threshold value. The number of relations in each class is too poor an indicator of correspondence between the design and code. A substantial improvement is achieved when the target class is added to each relation (the RC method). Nevertheless, the resulting plot is definitely at lower levels than all other methods.

Fig. 3 shows the recall values for each method over the analyzed components. As for Fig. 2, the points on each graph are joined through straight lines solely to help the visual comparison of the curves. Again the C and CA methods are associated with the highest curves, but now the difference with A and M is not that large, thus suggesting that methods based on class names and methods based on attributes can retrieve similar amounts of traceability information. The curve for the F method is much worse than that for the M method, but the combined effect on the recall is positive on several components—while, on the contrary, it was negative on precision performances. The shapes of the curves for the D and WD methods, the values of which are generally high, are similar. The result of the former is slightly improved by the latter, with two remarkable exceptions (components 13 and 16), on which weighted dictionaries

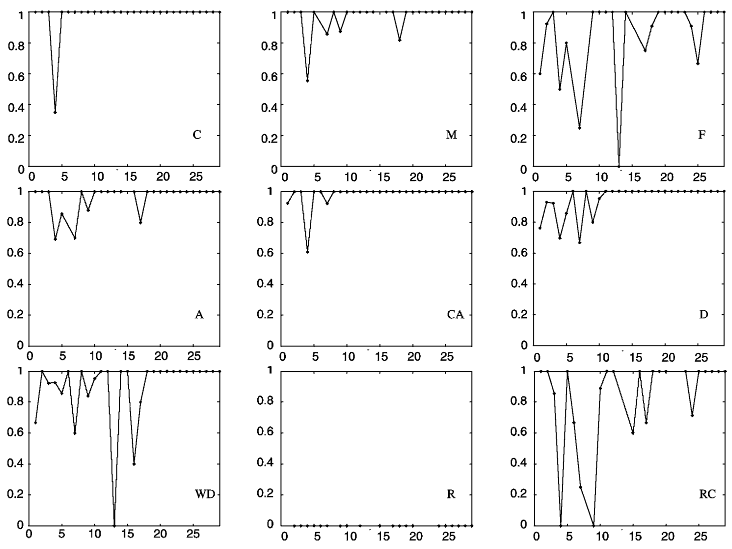

Fig. 3. Recall of different traceability recovery methods over all the analyzed components (on the horizontal axis).

provide a totally inefficient retrieval ability, lowering the recall parameter down to 0 and 0.4, respectively. The reason for such a performance decrease is the presence of infrequent words, associated to high weights, that are matched incorrectly.

The method based on the relations in each class cannot identify any of the class pairs to be matched. The reason was explained above, and consists in the impossibility to separate the distribution of the matched classes from the unmatched ones. An improvement can be observed in RC, but the overall performances, compared to the remaining methods, are poor.

Table 2 gives average precision and recall levels, computed over all the analyzed components, for the different traceability recovery methods. Such results confirm from a global perspective the detailed considerations made for the precision and recall curves over all the analyzed components.

CA is the method with the highest precision and recall, though the differences with the C method are minimal. The F method has lower overall performances than the M method, and their combination (method A) makes precision worse, while leaving substantially unchanged the overall average recall value. The D and WD methods do not exhibit relevant differences on precision. The slight decrease in the recall of the WD method is due to components 13 and 16, on which a dramatic worsening is produced by the introduction of the weights. Precision is not available (n/a) for the R method, which is not able to classify any class pair as matched, while its recall level is 0. Improvements are obtained by considering the target class of each relation, but the average values remain lower than those offered by the other methods.

The last two columns in Table 2 contain the average proportion of classes having no property among those required by each traceability recovery method. This case occurs,

Table 2
Average precision and recall of the different traceability recovery methods over all the analyzed components

Method	Precision	Recall	No info	
			Design	Code
C	0.971	0.977	0.0	0.0
M	0.889	0.966	0.298	0.004
F	0.727	0.852	0.398	0.269
A	0.872	0.960	0.270	0.002
CA	0.975	0.981	0.0	0.0
D	0.859	0.951	0.0	0.0
WD	0.846	0.895	0.0	0.0
R	n/a	0.0	0.157	0.363
RC	0.534	0.818	0.157	0.363

for example, when the design of a class does not contain the indication of any class field or method (actually, it is optional), so that the M and F methods do not have the basic anchor information available, and thus cannot determine any traceability link. The methods that suffer from this problem are those which exploit optional class properties; M needs the method names; F needs the fields; A needs both method names and fields. The R and RC methods require that class relations be specified. All the other methods use the class name, possibly combined with other information, which is always provided both in the design and code.

The high fraction of classes for which fields are unspecified in the design explains the poor performances of the F method. The situation is worsened by the high fraction of classes in the code containing no field. Such stateless classes are mere containers of functions (many of them are actually CORBA interfaces), or inherit the fields from a superclass and add new operations but no new fields. Methods unspecified in the design are a relevant fraction of the total, while there are very few classes in the code which contain only fields. This case occurs when a class defines data structures with public data fields and without operations. The availability of more information for the M method than for the F method (the difference is extremely high in the code), is the main reason for the superior performances of the M method. The proportion of classes without information diminishes within the A method, with respect to both the M and F methods, thus indicating that there is a compensation effect (i.e., classes without fields have methods and classes without methods have fields). It can be noted that there is a small number of classes in the code without fields and methods. They are generated automatically by support tools (e.g., the CORBA IDL compiler), and then left empty and never used.

Relations are unspecified in a relevant fraction of design classes. Missing relations in the code classes depend on the limitations of the reverse engineering tool that recovers them from the code. In fact, only static relations between classes are collected by such a tool (e.g., the presence of a pointer field referencing an object of another class). When the relation is created dynamically, the tool does not report it. Moreover many

Table 3
True match coverage

Method	$\lvert TM \setminus M \rvert$	$\lvert TM \cap M \rvert$	$\lvert M \setminus TM \rvert$
C	34	275	13
M	96	213	52
F	194	115	51
A	93	216	62
CA	23	286	12
D	81	228	59
WD	78	231	76
R	309	0	0
RC	207	102	79

code classes, which apparently operate as stand alone, are actually communicating with other parts of the system through interprocess communication facilities, but such dynamic relations are not represented in the class diagram, and cannot be extracted easily from the code. The unavailability of information necessary to the R and RC methods contributes to the poor level of precision and recall.

4.3. Relative coverage of the methods

Table 3 provides information on the ability of each method to cover the *true* matches, i.e. those class pairs that were manually labeled as truly corresponding (used by the maximum likelihood classifier to compute the thresholds within the cross validation procedure). For each method, the cardinality of the set-difference between the set of true matches, TM, and the set of found matches, M, is reported in column 2 ($\lvert TM \setminus M \rvert$). The cardinality of the set-intersection and the set-difference between M and TM is reported in columns 3 ($\lvert TM \cap M \rvert$) and 4 ($\lvert M \setminus TM \rvert$), respectively. Low average recall for the method is indicated by high figures in column 2, while high figures in column 4 indicate low precision. Note that average values in Table 2 were computed by averaging over the analyzed components, so that the same result cannot be determined from the values in Table 3, based on the whole true match set, with no regard to individual components.

Results described in the previous section are confirmed substantially by the true match coverage. CA and C are the methods giving the largest coverage of true matches. The D and WD methods follow, with only a slight improvement obtained by weighting dictionary words. The F method has lower performances than the M method, due to the unavailability of the anchors it uses. The M method's intersection with the true matches is slightly improved by the A method. The method based only on the count of the different kinds of relations (R) cannot retrieve any matched class pair, while the addition of the target class (in the RC method) improves the results, that still remain quite unsatisfactory.

The ability of each recovery method to extract information that is complementary to the other methods was evaluated, with the purpose of measuring its relative coverage.

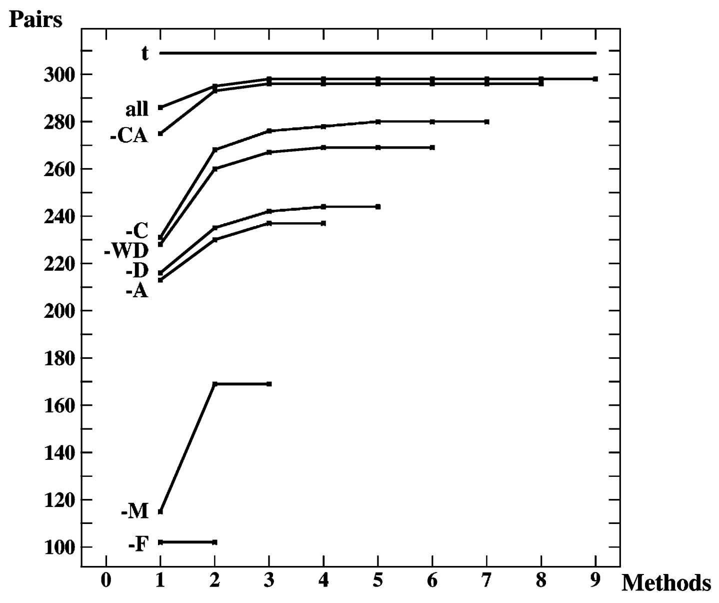

Fig. 4. Relative coverage of traceability recovery methods.

The class pairs recovered by a method could be a subset of the pairs recovered by another method, or could have a large intersection with that subset. As a consequence such a method does not enlarge its *coverage*, that is, the set of truly matched classes it retrieves.

A greedy comparison procedure was adopted to measure the differences in the relative coverage of methods. The method with the largest coverage was selected first. Then the method giving the largest coverage increment was considered, and the selection was repeated with the same criterion with all the remaining methods. Fig. 4 shows the coverage increase curve obtained with such a greedy selection procedure. Progressive coverage of the true matches set, *TM*, is reported as it was obtained starting from different combinations of methods. The horizontal segment at the top, labeled with **t**, marks the reference cardinality of *TM*. The curve labelled with **all** shows the progressive coverage of *TM* resulting from the application of the greedy procedure described above, starting from the whole suite of methods. Curves below that one are obtained by applying the same procedure, but excluding from the starting combination of methods the one having previously exhibited the highest coverage. So, for example, the curve labeled with -CA shows the progressive coverage of *TM* as obtained by excluding the CA method from the complete suite of methods; curve labeled with -C is obtained by further taking out the C method, and so on. Curve labeled with -F finally shows the poor coverage performances resulting from the sole RC method. The R method does not contribute to the improvement of any method combination, since it cannot retrieve any true match. It is responsible for the final horizontal (perfectly flat) segment of every line in Fig. 4.

The curve associated to the relative coverage of *all* methods indicates that the best performing method, CA, leaves little room for improvement to the other methods. Consequently the resulting plot (**all**) is very flat, suggesting that residual errors are very few and difficult to correct by exploiting alternative class properties. If the CA method is excluded, the result of the best performing method can be improved slightly, since the second best method can detect some of the true matches missed previously. Such an effect is even more evident if the best method of this curve (C) is in turn excluded. The starting point for the associated curve, marked with -C, is the WD method, which is highly improved by the second best method, in terms of additional true matches discovered. The last method to be excluded is F, leaving only RC and R in the set of methods to be combined. Thus the results presented in the previous section, indicating the F and RC methods as those with very poor performances (apart from R which cannot actually retrieve *any true match at all*), and the CA and C methods as the best ones, are confirmed.

5. Discussion

The results presented in the previous section are based on design and code information gathered from just one work environment, in which specific procedures and habits are adopted. However, there are reasons to believe that some of the outcomes can be generalized reasonably and that lessons of a wider interest can be learned from them. In a specific working environment, the case may well occur that the relevant linkage properties differ from the ones emerging here. The method illustrated still provides an empirical way of assessing the relevance of linkage properties in different situations.

The strongest hypothesis that should be satisfied by a work environment so that the following considerations apply is that *traceability is not explicitly supported*. In fact, if a tool is available that explicitly records traceability links and automatically extracts the code entities associated to a design, there is no need to rely on the preservation of any property. Even the class name can be completely different between design and code, since it is the tool that takes care of keeping them linked. This is, for example, the case of the design tool Rose [16], provided it is used properly.

If traceability is supported implicitly, the preservation of class properties when moving from design to code depends on the way programmers read, understand, elaborate and finally implement the architecture that is delivered to them by the designers. Commonalities among different environments can be devised on class name, class structure and class dictionary.

Traceability recovery methods based on **class name** are the best performing. In the authors' opinion, this is not an accident of the analyzed data; it corresponds to the identity determination and semantics attribution principles. The class name is the unique fundamental identifier of classes both in design and code. Class names are referenced each time an object of that type is created or a relation with that class is built. When transforming design into code, class identity is preserved primarily by maintaining the main class identifier (i.e., the class name). Looking more closely at our data, indeed more than 1/4 of the class names change when moving from the design to the code; for a large majority of the cases, however, changes consist of just adding a prefix (about 1/3 of the cases) or of small “refinements”, such as capitalization or the contraction/expansion of words. There is an additional reason why the class name remains so stable: it is also an extreme summary of the whole class semantics, meant as the domain or programming concept it embodies. When a software engineer needs to understand the behavior of an object-oriented system, the class names are an initial precious source of information. They are not mere identifiers, that could be replaced by automatically generated symbols. Rather, they carry relevant information on the core ideas behind the class. Semantics preservation is a main objective of the refinement of design into code, so that class names are expected to be extremely stable.

On the contrary the **class structure** (i.e., the net of relations with other classes) is only indirectly associated with the class identity; it represents the mutual dependencies and linkages among classes. Class relations provide useful suggestions on the kind of services a class needs from other classes or exports toward the external world, but the same pattern of relations can be replicated several times with a totally different meaning. Therefore the class structure seems to carry little information on identity and too coarse-grained information on semantics, resulting in a poor traceability indicator. There are additional reasons why the class structure is not expected to produce good traceability methods. First of all, the class structure that can be retrieved from the code through reverse engineering is a very raw approximation of the intended structure. For example, aggregations and associations can be hardly distinguished. Being able to retrieve accurate design representations from the code is an open research field, whose usefulness goes beyond traceability. Moreover, the only information surely attached to a given relation is the relation type, since relation name and roles are optional. As a consequence the basic properties used to recover traceability links carry a very small amount of useful information.

Class dictionaries are obtained by segmenting the class name and class attribute identifiers into individual words that stand alone, thereby eliminating each word’s place in the identifier. The sequential structure of the words that make up a class identifier is not random, but rather reflects some sort of *syntax* (see [6]) that is respected by designers and programmers when building new identifiers, to make them readable and meaningful. Breaking the identifiers into single words implies that the syntactic structure of the identifiers is totally ignored. The lower performances of methods based on dictionaries, with respect to those that do not break the identifier structure, indicate that there is a relevant loss of information in the segmentation process. The presence of a set of words in a class is not such a strong traceability indicator as the presence of the same words in an ordered sequence within an identifier.

Some final considerations can be made on the overall approach to traceability recovery. Using the right basic anchors is important, but another relevant issue is the ability to address spurious information in design and code, for example, the library or external classes reported in the design for clarity, the code automatically generated by support tools (GUI builders, etc.), and the classes which are part of COTS (commercial off-the-shelf components). Such sources of *noise* are expected to be found in many real work environments, and must be dealt with properly in order to recover traceability links accurately. In this work, the use of a maximum match algorithm followed by a maximum likelihood classifier helped to distinguish noisy pairings of classes from the true ones with good performances in terms of precision and recall. Additional strategies to handle the inevitable noise could be necessary in a different context.

6. Related work

This paper complements previous works [1,2] in that the focus is on the basic class attributes to be used as traceability anchors, rather than the traceability procedure itself. Only one traceability method, namely CA, was considered in [1,2]. The advantages and disadvantages of its selection, with respect to the other analyzed alternatives, are clarified in this work. Moreover, the traceability recovery process may require a preliminary method evaluation similar to the one described here, when it is applied in a new industrial environment.

Among the works about model-implementation compliance checking that have been proposed in the literature (see, for example, [13–15,18,19]), here we concentrate on those closest to our approach (i.e., those that address explicitly the problem of checking design against implementation and are applicable in the object-oriented domain).

The work by Meyers et al. [14] differs from ours both in its objective and its implementation. The objective of CCEL is to check the compliance of a program against a set of design guidelines expressed as constraints that affect single or groups of classes, while our objective is to check the compliance of a design model against its implementation. Unlike CCEL, we have an explicit design model which states the existence of a set of specific entities with specific properties and relations among them, and this must be verified in the implementation.

The work by Murphy et al. [15] is much closer to ours. Software reflexion models can be applied in the OO domain: Murphy et al. refer to an experiment on an industrial subsystem where a reflexion model was computed to match a design expressed in the Booch notation against its C++ implementation. Their process and ours are similar and many analogies can be drawn. We both use an extraction tool to derive abstract information from source code. The reflexion model tool is analogous to our design-code matcher, in that they both provide an output in terms of where the high-level model agrees or disagrees with the model of the source code. Where their and our approach differ mainly is in the use of the mapping between the two models. They use mapping to trace the source code model entities onto the high-level model entities. Yet, the nature and granularity of the two models are quite different. For example, they have modules in the high-level model and functions in the source code model: the mapping information is used to cluster the source code model entities in order to assign them to the high-level model entities. For this purpose they make use of regular expressions and exploit naming conventions of source code entities. In our case, the entities of the two models are exactly the same: classes and relations among them, and matching is based on the similarity of properties, thus allowing a *partial* matching between entities in the two models. In fact, coding standards, naming conventions and programming style may alter design names to accommodate implementation details or shortcuts.

Pattern-Lint, the system developed by Sefika et al. [19], differs from our work and Murphy's, which are based only on static analysis, in that it also integrates dynamic visualization. Its results are then compared to the static-analysis ones. Although it is a very general and powerful framework, Pattern-Lint is able to check the compliance of source code with respect to three types of design models: coding guidelines, architectural models such as design patterns or styles, and heuristic models such as coupling and cohesion. Pattern-Lint differs from our work mainly in that the tool is not provided with a high-level model representing the entire system directly in terms of classes and relations that is later compared with the corresponding information in the source code. Higher-level models or partial models, which represent pieces of a system, are provided to check the compliance with specific parts of an implementation. Moreover, Pattern-Lint does not handle approximate matches like our system does using similarity measures.

7. Conclusion

Different design-code traceability methods have been presented, based on different class properties or property combinations. The anchors for traceability recovery are those atomic class properties that are expected to be persistent when the design is refined into the code. Examples include the class name, the attributes, the dictionary employed by a class and its relations.

An experimental benchmark consisting of 29 industrial components was available for the evaluation of the performances of the proposed traceability methods. For each of them, both design and code were analyzed and traceability links between class pairs were recovered. The two basic performance indexes used are recall and precision. The fraction of truly matching class pairs retrieved by each method provides its recall, while precision is its ability not to classify as matched classes that are actually unmatched. A true match coverage figure was also computed, while the relative independence of the methods was measured by considering the coverage increment that each of them produces. All these results suggest that the best performing methods are those with an explicit representation of the class name, while very poor performances are associated to methods based on the relations of a class with the other classes. Intermediate results are achieved by dictionary-based methods.

Although this work is based on data from a single industrial environment, some of the outcomes can be generalized reasonably. The class name is an important traceability indicator, being associated with the identity of the class looked for in design and code, and being also a powerful and concise way to express the class semantics informally.

On the contrary, class relations are associated to the class identity only indirectly and provide limited information on semantics. Class dictionaries contain all the needed semantic information about a class, but the syntactic structure is lost when identifiers are segmented into individual words.

Future work will be devoted to evaluating the impact of choosing the edit distance as a means to determine the similarity between string type class properties. Different traceability methods may be affected in a different way by such a choice. In particular, methods based on class dictionaries could employ a synonym dictionary instead. Another direction for future investigation is the analysis of the syntactic structure of the class identifiers, which carries important traceability information.

References

[1] G. Antoniol, B. Caprile, A. Potrich, P. Tonella, Design-code traceability for object-oriented systems, Ann. Software Eng. 9 (2000) 35–58.

[2] G. Antoniol, A. Potrich, P. Tonella, R. Fiutem, Evolving object-oriented design to improve code traceability. Proc. 7th Internat. Workshop on Program Comprehension, Pittsburgh, Pennsylvania, USA, May 5–7, IEEE Computer Society, 1999, pp. 151–160.

[3] I.D. Baxter, C.W. Pidgeon, Software change through design maintenance, Proc. Internat. Conf. on Software Maintenance, Bari, Italy, October 1–3, IEEE Computer Society, 1997, pp. 250–259.

[4] M. Bunge, Treatise on Basic Philosophy: Vol. 3: Ontology I: The Furniture of the World. Reidel, Boston, MA, USA, 1977.

[5] M. Bunge, Treatise on Basic Philosophy: Vol. 4: Ontology II: A World of Systems, Reidel, Boston, MA, USA, 1979.

[6] B. Caprile, P. Tonella, Nomen Est Omen: Analyzing the Language of Function Identifiers, Proc. 6th Working Conf. on Reverse Engineering, WCRE'99, October 6–8, IEEE Computer Society, Atlanta, GA, USA, 1999, pp. 112–122.

[7] S.R. Chidamber, C.F. Kemerer, A metrics suite for object-oriented design, IEEE Trans. Software Eng. 20 (6) (1994) 476–493.

[8] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, USA, 1990.

[9] R.O. Duda, P.E. Hart, Pattern Classification and Scene Analysis, Wiley, New York, NY, USA, 1973.

[10] E. Gamma, R. Helm, R. Johnson, J. Vlissides, Design Patterns: Elements of Reusable Object Oriented Software., Addison-Wesley Publishing Company, Reading, MA, USA, 1995.

[11] D. Garlan, M. Shaw, Software Architecture: Perspectives on an Emerging Discipline, Vol. 1, Prentice-Hall, Englewood Cliffs, NJ, USA, 1996.

[12] M. Lindvall, K. Sandahl, Practical implications of traceability, Software: Practice Exp. 26 (10) (1996) 1161–1180.

[13] D. Luckham, F. von Henke, B. Krieg-Bruckner, O. Owe, in: Anna, a Language for Annotating Ada Programs: Reference Manual, Lecture Notes in Computer Science, Vol. 260, Springer, Berlin, Germany, 1987.

[14] S. Meyers, C.K. Duby, S.P. Reiss, Constraining the Structure and Style of Object-Oriented Programs. (CS-93-12), Brown University, Providence, RI, USA, 1993.

[15] G.C. Murphy, D. Notkin, K. Sullivan, Software reflexion models: bridging the gap between source and high-level models, Proc. 3rd ACM Symp. on the Foundations of Software Engineering, Washington, DC, USA, October 12–15, 1995, ACM Press, New York, 1995, pp. 18–28.

[16] Rational Software Corporation, Rational Rose 98, Using Rose, Rational Software Corporation, Cupertino, California, USA, 1998.

[17] J. Rumbaugh, M. Blaha, W. Premerlani, F. Eddy, W. Lorensen, Object-Oriented Modeling and Design, Prentice-Hall, Englewood Cliffs, NJ, USA, 1991.

[18] R.W. Schwanke, An intelligent tool for re-engineering software modularity, Proc. 13th Internat. Conf. on Software Engineering Austin, TX, USA, May 13–17, 1991, IEEE Computer Society/ACM Press, New York, 1991, pp. 83–92.

[19] M. Sefika, A. Sane, R.H. Campbell, Monitoring compliance of a software system with its high-level design models, Proc. 18th Internat. Conf. on Software Eng, Berlin, Germany, March 25–29, 1996. IEEE Computer Society Press, Los Alamitos, CA, USA, 1996, pp. 387–396.

[20] M. Stone, Cross-validatory choice and assessment of statistical predictions (with discussion), J. Roy. Statist. Soc. B 36 (1974) 111–147.