Revisiting the Delta IC Approach to Component Recovery

Gerardo Canfora[†], Jörg Czeranski[‡], and Rainer Koschke[‡]
[†]*University of Sannio, Palazzo Bosco Lucarelli, Piazza Roma, 82100 Benevento, Italy*
[‡]*University of Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany*
gerardo.canfora@unisannio.it, {czeranski, koschke}@informatik.uni-stuttgart.de

Abstract

Component recovery supports program understanding, architecture recovery, and re-use. Among the best known techniques for detection of re-usable objects (related global variables and their accessor functions) is Delta-IC [2]. This paper re-visits the original approach and extends it in different ways. It describes a variant of Delta-IC suitable for reverse engineering that omits the slicing step of the original approach. The underlying metric of Delta-IC is extended toward types integrating ideas of the Internal Access technique [32] such that abstract data types can also be detected. Furthermore, the connectivity metric of Delta-IC is combined with a cohesion metric based on vertex connectivity. The new metrics and the new algorithm for reverse engineering are quantitatively evaluated using the framework proposed in [19] as a standard evaluation of clustering techniques for component recovery.

1. Introduction

Recovering modules and subsystems from existing software systems has proven useful in a number of ways and many methods to automatically or semiautomatically detect components have been published in the literature [1, 2, 5, 6, 8, 10, 12, 14, 15, 16, 17, 21, 23, 24, 25, 26, 27, 28, 29, 30, 32]. The abundance of published methods calls for frameworks to unify, classify, and compare them in order to make informed decisions. Girard and Briand introduce a process which synthesizes many methods to extract components from code [11]. Lakhotia [20] and Koschke [18] present a comprehensive overview and a classification of existing component recovery methods; Girard and Koschke compare six published methods which recover abstract data types and objects [13]. Koschke and Eisenbarth discuss the need for a standardized approach to compare component recovery methods and propose a framework to cost-effectively run quantitative evaluation experiments [19]. The framework evolved from a quantitative evaluation of diverse methods which detect the four kinds of components described in Table 1 [13, 18].

One of the methods evaluated in [18] is the connectivity-based *Delta-IC* approach defined by Canfora et al. [2]. In its original formulation, the method detects abstract data objects (ADO), but it can be extended to detect abstract data types as described in Section 4.2. Basically, the method consists of two parts (see Figure 1):

1. A fully automatic technique that identifies ADOs in the form of clusters of global variables and subprograms that set and use them.
2. A repeated application of the automatic technique intertwined with human validation and the application of slicing to subprograms that are part of more than one cluster.

Table 1. Types of components.

An *abstract data object* (ADO) is a group of global variables and constants together with the routines which access them.
An *abstract data type* (ADT) is an abstraction of a data structure (a user-defined type) and all the type's valid operations on that data structure.
A *hybrid component* (HC) is an abstract data type that uses global variables to save state information. For example, an implementation can count in a global variable how many instances of the ADT have been created.
A set of *related subprograms* (RS) are subprograms that together perform a logical function, i.e., have functional cohesion [33].

```
repeat
  build refer-to graph
  for each subprogram S loop
    if ΔIC(S) ≥ Θ then
      if candidate-cluster (S) is accepted by user then
        collapse ref-by (S) into a single new variable node
    else
      slice S using different variables of candidate-cluster (S)
until graph contains only isolated subgraphs consisting of a
      variable grouping with one or more functions
```

Figure 1: Original Delta-IC approach.

The fully automatic technique exploits a graph, called a *refer-to graph*, in which nodes are either subprograms, global variables, or constants and edges connect subprograms to the global variables and constants they reference. The technique relies on a metric called *internal connectivity* (IC) that measures the fraction of the edges internal to a cluster with respect to the total number of edges that have at least one vertex in the cluster. *IC* captures the coupling of a candidate ADO with respect to the rest of the system.

The evaluations presented in [13] and [18] used a slight-

ly different version of the method. The first difference is that only the automatic technique is applied and the second is that a two-step process replaces the iterative nature of the method. At first, global variables, constants, and subprograms are clustered to ADOs according to the automatic technique. Then, all resulting clusters are rejected whose internal connectivity is below a given threshold. These differences are mainly due to the different focus of the work described in [2], which is aimed at finding reusable components and therefore involves changing the original code, and the research discussed in [13] and [18], whose scope is reverse engineering for program understanding and therefore excludes changing the original code through slicing. The differences are also the consequence of a choice made in [13] and [18] to compare only fully automatic component recovery methods.

Overview. This paper revisits the IC-based method to extend it in two different directions, namely, recovering ADTs and finding a better cohesion metric to identify the candidates. The remainder of the paper is organized as follows. Section 2 reviews related research, sets the original contribution of this paper, and introduces the terminology used throughout the paper. Section 3 recalls basic definitions of the IC-based method and discusses its properties. Section 4 introduces the new method for reverse engineering combined with alternative cohesion metrics, and Section 5 provides quantitative results of applying the new variants.

2. Related Research

There exist several approaches to automatic ADO and ADT recovery within procedural programs. Referring to the categorization proposed in [17] they can be grouped into domain-model-based approaches (e.g. [8, 9]), data-flow-based approaches (e.g. [30]) and structure-based approaches. The latter family of approaches has been the most widely investigated in the past years and there is a fair amount of methods reported in the current literature (for a summary, see [18]). Structure-based approaches can be further categorized into connection-, metric-, graph-, and concept-based approaches.

Connection-based approaches exploit direct relationships between code entities, such as subprogram signature types and accessed variables, to define the clusters. Examples of connection-based approaches include the *Same Module* technique [14], the *Part Type* technique [25], and the *Internal Access* technique [32].

Metric-based approaches are iterative in nature and use metrics to determine the clusters. Schwanke's method [28] uses a similarity metric derived from direct calls among subprograms and usage of non-local entities; Girard, Koschke, and Schied [12] improve the method by distinguishing between different uses of non-local entities. The type-based method [26] uses a similarity metric that counts the portion of types of parameters and local and non-local variables of subprograms.

Graph-based approaches are similar to connection-based approaches, but the substantial difference is that they exploit graph-theoretic analyses on the whole graph to define the clusters. *Dominance* analysis [4] and *Strongly Connected Components* analysis [4] are examples of graph-based approaches.

Finally, mathematical concept analysis has been proposed as a method to identify ADOs and ADTs [3, 21, 27, 29]. Concept analysis is more general than graph-based and connection-based methods as it can capture the same kinds of relationships depicted in graphs and presents several additional advantages. These include a finer control over the granularity of the obtained modularization and an improved discriminatory power: a concept lattice defines a range of modularization options that can be chosen for different situations.

The original *Delta-IC* method [2] combines features of the connection-based and metric-based approaches. It first generates clusters using connection information, mainly the accesses of subprograms to global variables, and then exploits a metric to filter the generated clusters. This is a distinctive feature of the method, as other connection-based approaches simply cluster without rating the generated clusters.

2.1. Contributions

The original *Delta-IC* approach described by Canfora et al. aims at finding reusable components [2]. Consequently, the system may in fact be changed in this kind of application. This paper describes a variant of *Delta-IC* useful for reverse engineering in which changes are not allowed and therefore program slicing is not possible. Thus, overlapping candidates can result. The *Delta-IC* variant for reverse engineering deals with overlapping components by merging very similar candidates, yet overlapping candidates may remain if the components are not similar enough. In the following, the term *Delta-IC* will refer to the variant for reverse engineering if nothing else is said.

The original *Delta-IC* metric is based on variables and subprograms referencing these variables and is therefore able to identify abstract data objects only. This paper will extend the definition of *Delta-IC* to types and their relationships to subprograms in order to identify abstract data types.

The quantitative evaluation of *Delta-IC* published in [18] used a predecessor of the evaluation framework described in [19]. This paper re-evaluates *Delta-IC* and relat-

ed fully automatic techniques by the new evaluation framework proposed in [19] using the benchmarks described in [18]. As a by-product of the evaluation in [18], the analysis of available data suggested that the effectiveness of the method could be improved by ignoring the part of *Delta-IC* that measures cohesion of candidates, i.e, taking into consideration only the coupling of candidates according to *IC*. This interesting outcome motivated our search for a better cohesion metric. One intuitive way to capture cohesion of ADOs is by way of the so-called vertex connectivity of a graph: A graph has *vertex connectivity K* if the deletion of any *K*-1 nodes fails to disconnect the graph [7]. In this paper, we investigate alternative integrations of the original coupling measurement of *Delta-IC* with the vertex connectivity as a cohesion metric and evaluate these alternatives by application to the same set of systems used in the evaluation [18].

2.2. Terminology

A *component* is a set of related entities that together have either functional [33] or abstract cohesion [22]. A *cluster* is a set of subprograms (functions and procedures), variables, constants, and user-defined types proposed as a candidate component. The *elements* of a component or cluster are all entities contained in this component or cluster, respectively. A subprogram, *S*, is said to reference a variable, *V*, if *S* sets or uses *V* or if *S* takes the address of *V*, denoted by *reference*(*S, V*) (until Section 4.2, we consider the relation *refer-to*(*S,V*) a synonym of *reference*(*S,V*); in Section 4.2, *refer-to* will be generalized). The set of referenced variables of a subprogram *S* is *referenced-variables*(*S*) = {*V* | *reference* (*S,V*)}. Inversely, the set of subprograms referencing variable *V* is *referencing-subprograms* (*V*) = {*V* | *reference* (*S,V*)}. A type *T* mentioned in the signature of a subprogram *S* is said to be a signature type, denoted by *signature-type*(*S, T*).

3. Delta IC Approach

High cohesion in the case of an abstract data object *S* implies that each of the subprograms in *S* references many variables of *S*; low coupling implies that each of the subprograms of *S* references only very few variables that do not belong to *S* and that only few subprograms from outside of *S* reference variables of *S*. The approach proposed by Canfora et al. is heading in this direction. It basically consists of two parts. At first, variables and subprograms are clustered to ADOs according to a specific usage pattern. Then all resulting clusters are rejected whose internal connectivity is below a given threshold. The internal connectivity metric proposed by Canfora et al. is described below.

3.1. Original Definition

The clustering pattern and the evaluation metric are defined on the refer-to graph that describes the usage of global variables and constants by subprograms (in the following, the term *variable* always refers to global variables and constants). They can be explained more easily in terms of the following definitions, given a subprogram *S* and a global variable *V*:

subprograms related to *S* are all subprograms which reference variables also referenced by *S*:

$$\text{subprograms-related-to (S)} = \bigcup_{e \in \text{ ref-by (S)}} \{F \mid F \in \text{ refer-to(e)}\} \quad \textbf{(1)}$$

where *refer-to (e) = referencing-subprograms (e)* and *ref-by* (*S*) = *referenced-variables (S)*.

The reason why *refer-to*/*ref-by* are introduced here — instead of using *referencing-subprograms/referenced-variables* directly — is that *Delta-IC* will be extended to types by re-defining *refer-to* and *ref-by* in Section 4.2.

closely-related subprograms of S are all subprograms which reference ***only*** referenced variables of S:

$$\text{closely-related-subprograms } (S) = \bigcup_{e \in \text{ ref-by } (S)} \{F \mid F \in \text{ refer-to}(e) \wedge \text{ref-by}(F) \subseteq \text{ref-by}(S)\} \quad \textbf{(2)}$$

Example. Given the refer-to graph of Figure 2 and *F* as the subprogram under consideration, then the variables *ref-by(F)* are {v_1, v_2}, the *subprograms related* to *F* are {F, f_1, f_2, f_3}, and the *closely related subprograms* of *F* are {F, f_1, f_2}.

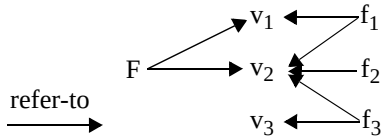

Figure 2: Example refer-to graph.

The candidate that is proposed as an abstract data object consists of all *closely related subprograms* of the given subprogram *S* plus the variables *referred by S*:

$$\text{candidate-cluster (S)} = \text{closely-related-subprograms (S)} \cup \text{ref-by (S)} \quad \textbf{(3)}$$

Example. In the example of Figure 2, the candidate cluster is {v_1, v_2, F, f_1, f_2} for the given subprogram *F*. Note that the proposed clusters depend upon the given subprogram. Suppose *F* also referenced variable v_3, then the cluster for *F* would be {v_1, v_2, v_3, F, f_1, f_2, f_3}; from the perspective of f_3 we would get the cluster {v_2, v_3, f_2, f_3}. Thus, clusters can overlap.

The candidate cluster is ranked by the internal connec-

tivity metric and only proposed if this metric yields a value greater than a user-determined threshold. The *internal connectivity* measure (IC) and the *improvement in internal connectivity* (ΔIC) are defined as:

$$IC(S) = \frac{\sum_{e \in \text{ref-by (S)}} |\{F | F \in \text{refer-to}(e) \wedge \text{ref-by}(F) \subseteq \text{ref-by}(S)\}|}{\sum_{e \in \text{ref-by (S)}} |\{F | F \in \text{refer-to}(e)\}|} \quad \textbf{(4)}$$

$$\Delta IC(S) = IC(S) - \sum_{e \in \text{ref-by (S)}} \frac{|\{F | \text{ref-by (F)} = \{e\}\}|}{|\text{refer-to } (e)|} \quad \textbf{(5)}$$

IC(S) is the portion of references to individual variables of the cluster from subprograms also inside the cluster (closely related subprograms) with respect to the number of all references. If there is no reference from outside the cluster, IC(S) is 1. In the example of Figure 2, IC(F) is as follows: (2 + 3) / (2 + 4) = 0.83. The subtrahend in the definition of ΔIC reflects the portion of subprograms that reference only a single variable of the cluster with respect to the number of all references to that variable. In the example of Figure 2, the subtrahend of ΔIC is 1/4: f_2 is the only subprogram that accesses a single variable only, namely, v_2, which is referenced by 4 subprograms. Consequently, ΔIC(F) = 0.83 - 0.25 = 0.58.

The underlying intuition of the definition of IC is to have only few references of variables from outside the cluster. The motivation for ΔIC will be discussed below in more detail.

The original *Delta-IC* approach uses the clustering algorithm in Figure 1 [2] where the proposed cluster for a given subprogram *S* is *candidate-cluster*(*S*) if ΔIC (*S*) ≥ Θ. It may be a sign of loose relatedness when a candidate's internal connectivity is below the threshold. The reason may be that the subprograms implement distinct logical functions and therefore reference unrelated variables. The code of such subprograms could be separated into distinct parts that correspond to the distinct logical functions by means of program slicing [31]. This is what Canfora et al. proposed in [2].

3.2. Properties of the ΔIC Definition

The definition of ΔIC consists of two parts. The subtrahend in (5) covers substructures of the candidates that consist of only one variable and those subprograms that access solely this variable. The original motivation for the subtrahend in the definition (5) of ΔIC is to measure the "improvement in internal connectivity". Recall the iterative nature of the algorithm in Figure 1: In each step, the variables of an accepted candidate are collapsed into a new single variable that serves as a representative of the cluster further on. For example, if the candidate cluster around *F* in Figure 2 is accepted, the refer-to graph in Figure 3 results.

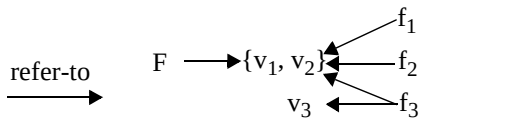

Figure 3: Collapsed refer-to graph.

In the second iteration, there is actually an accepted candidate {F, f_1, f_2, v_1, v_2} whose ΔIC is now 3/4 - 3/4 = 0. If f_3 is chosen, there are the three functions of the accepted cluster that now access only one single variable - the collapsed node - and hence ΔIC (f_3) = 1 - 3/4. The increase of internal connectivity for the accepted cluster in the second iteration is therefore 1/4.

The subtrahend of ΔIC is motivated by the fact that the clustered variables are collapsed. Yet, in the first iteration, there are no clustered variables, i.e., each variable stands for itself. In that case, the subtrahend actually represents the internal connectivity of clusters around a single variable that consist of subprograms that only access this variable and no other variable, i.e., if C_v is a cluster that consists of a single variable *V* and all subprograms that only refer to this and no other variable, then the following equation holds (a proof can be found in [18]):

$$\forall S \in C_V: \; IC(S) = \sum_{e \in \text{ref-by (S)}} \frac{|\{F | \text{ref-by (F)} = \{e\}\}|}{|\text{refer-to } (e)|} \quad \textbf{(6)}$$

Such substructures around a single variable might be considered a candidate on their own and therefore it could make sense to subtract their internal connectivity from the overall internal connectivity of the composite structure. Yet, this is intuitively not appropriate for the following reasons:

- The decision to consider only subclusters of single variables is arbitrary. Why not considering subclusters with two or more variables?
- Furthermore, one should think that a subprogram that references one variable only and this variable is in the cluster, the subprogram should definitely also be in the cluster. An example is an abstract data object *stack* based on two global variables *stack_content* (array for the stack content) and *stack_pointer* (index into *stack_content*) having an accessor function *size* to return the number of elements on the stack; *size* would need to reference *stack_pointer* only and still does belong to the cluster.

These intuitive counter arguments are confirmed by the quantitative evaluation described in Section 5. Consequently, the sum of the subtrahend for ΔIC should run over collapsed variables only.

A corollary of (6) is that for a cluster consisting of a sin-

gle variable, *V*, and its accessor functions $S_1,\ldots,S_n$ ($n \geq 1$) that only reference *V*, $\Delta IC(S_i) = 0$ holds for all S_i ($1 \leq i \leq n$). Note that this does not depend upon whether information hiding is employed, i.e., whether there are other subprograms from outside the cluster that reference the cluster variable. In other words, the metric fails to make a distinction for coupling of clusters with a single variable. This may be particularly problematic when the metric is extended to types because abstract data types mostly consist of a single type only.

Another problematic property of ΔIC is that its two constituents are unbalanced: While the value of *IC* can only be between 0 and 1, the subtrahend of ΔIC can be between 0 and *n* (where *n* is the number of variables in a cluster). Given a subprogram, *S*, that refers to variables $V_1,\ldots,V_n$ where each variable V_i in $V_1,\ldots,V_n$ is accessed by *m* other subprograms $S_{i,1},\ldots,S_{i,m}$ and $ref\text{-}by(S_{i,j}) = \{V_i\}$ for $1 \leq j \leq m$ (see Figure 4). Then, the subtrahend of ΔIC is as follows:

$$\sum_{e \in \text{ ref-by}(S)} \frac{|\{F \mid \text{ref-by (F)} = \{e\}\}|}{|\text{refer-to } (e)|} = n \times \frac{m-1}{m} \quad \text{where}$$

$\frac{m-1}{m}$ approaches 1 for large *m*, hence, $\Delta IC(S) \approx 1 - n$.

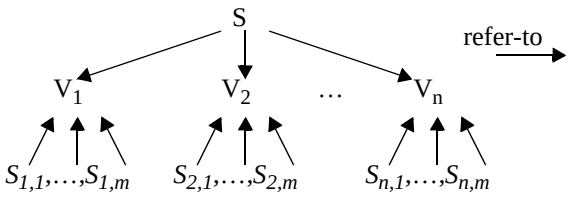

Figure 4: Cluster with low ΔIC.

Furthermore, the metric considers only coupling but not cohesion of the candidate clusters though one would expect that both coupling and cohesion should be taken into account.

A minor point of critique is that the term *internal connectivity* is somewhat misleading. What the formula for the internal connectivity measures is the fraction *closely-related-subprograms* versus *related-subprograms* with respect to individual variables of the cluster, which is a relationship between the cluster and its environment as opposed to an internal property.

4. Extensions

In a pure reverse engineering process for program understanding, the system must not be changed (it may be changed afterwards), i.e., physically slicing subprograms whose ΔIC value is below a threshold is inappropriate. Leaving out slicing reduces the loop in Figure 1 to one iteration and overlapping candidates may remain. Merging these overlapping candidates regardless of the degree of overlap is not satisfactory. This approach was taken in an earlier evaluation of *Delta-IC* [13] in order to get a fair evaluation since other methods of this evaluation always produce distinct candidates. For the application of the *Delta-IC* method, we can do better: We can merge two candidates when they share a large amount, otherwise they remain distinct and overlapping. In particular, this is the right approach when the user can be consulted. Merging similar candidates frees the user from an overwhelming number of similar candidates: She or he has to judge only critical cases.

4.1. Extended Algorithm for Reverse Engineering

The *Delta-IC* algorithm for reverse engineering in Figure 5 merges candidates only when they have many elements in common (Step 3). We treat one component *S* as a part of another component *T* when $S \subseteq_p T$ holds according to the following partial subset relationship, $\subseteq_p$, which allows for inexact matches:

$$S \subseteq_p T \text{ if and only if } \frac{|\text{elements}(S) \cap \text{elements}(T)|}{|\text{elements}(S)|} \geq p \quad \textbf{(7)}$$

where $0.5 \leq p \leq 1.0$ is a tolerance parameter *p* that needs to be specified for the comparison. If set to 1.0, *S* must be completely contained in *T*. A more pragmatic adjustment is $p = 0.7$, i.e., at least 70 percent of the elements of *S* must also be in *T*. This number is motivated by the assumption that at least three elements of a four-element component must also be in the other component to be an acceptable match. Step 3 merges the overlapping candidates. For the connectivity metric, *conn*, in this algorithm, ΔIC is used. Later we will propose alternative connectivity metrics, but the algorithm remains the same.

4.2. Generalization for Types

The internal connectivity metric was originally proposed only for abstract data objects. However, we can extend the domain of connectivity to types as well. Before we actually generalize the metric, we state some observations.

There are two different kinds of entities of an abstract data object: variables and constants that we do not want to be accessed from outside of the abstract data object and subprograms that act as public accessor routines. According to these two classes, there are the following different kinds of relationships that we implicitly distinguished above:

1. **non-abstract usage**: a variable is directly referenced by a subprogram. There are two categories of non-abstract usage:
   - **a.** the variable is non-abstractly used by a subprogram **within** the component

Input:

- refer-to graph G
- connectivity threshold Θ

Output:

- set of component candidates C

Algorithm:

1. *generate candidates:*
   for each subprogram S **in** G **loop**
   cluster (S) := candidate-cluster (S);
   end loop;
2. *filter candidates whose connectivity is less than* Θ:
   for each subprogram S **in** G **loop**
   if conn (S) < Θ **then**
   cluster (S) := ∅;
   end if;
   end loop;
3. *merge overlapping candidates:*
   while ∃ a pair of subprograms {S1, S2} **in** cluster
   where
   (cluster (S1) $\subseteq_p$ cluster (S2) ∨ cluster (S1) ≠ ∅)
   ∨ (cluster (S2) $\subseteq_p$ cluster (S1) ∨ cluster (S2) ≠ ∅)
   loop
   cluster (S1) := cluster (S1) ∪ cluster (S2);
   cluster (S2) := ∅;
   end loop;
4. *return results (filter trivial components)*:
   for S **in** cluster'Range **where** | cluster (S) | > 1
   loop
   C := C ∪ {cluster (S)}
   end loop;

Figure 5: Delta-IC for reverse engineering.

b. the variable is non-abstractly used by a subprogram **outside** of the component

2. **abstract usage**: a variable is not used directly by a subprogram *S* outside of the component but by an accessor routine of the component called by *S*, in other words: *S* is accessing the variable only by means of the accessor routine associated with the variable.

Cases 1.a and 2 conform to the information hiding principle, case 1.b does not. Hence, metrics for coupling should penalize 1.b. The metrics for variables and types in this section are defined with this in mind.

As opposed to variables, we do not want to hide types - they would not be of any use then. Instead, we want to hide the underlying data structure of a type. This corresponds to the idea of the *Internal Access* heuristic [32]. Types should be used abstractly by subprograms outside of the abstract data type. A non-abstract usage of a type *T* by a subprogram *S*, denoted by *non-abstract*(*S, T*), is as follows [18] (let *E* be an expression of type *T*):

- if *T* is a record, then any field selection *E* in *S* is a non-abstract usage of *T*
- if *T* is an array, then any index subscript *E* in *S* is a non-abstract usage of *T*
- if *T* is a pointer, then any dereference *E* in *S* is a non-abstract usage of *T*
- if *T* is a standard type, then any application *E* of a standard operator in *S* is a non-abstract usage of *T*

Now that we have a unifying concept *non-abstract usage* for both types and variables, we can generalize the specification of *refer-to* and *ref-by* accordingly. The formulas (1) - (5) need not be changed. So far, *refer-to(v)* has been defined as *referencing-subprograms(v)*. Hence, the definitions of *refer-to* and *ref-by* can be extended as follows in order to include the restricted signature-types relationships (only those signature types are considered that are tagged as non-abstract usage):

$$\text{refer-to}(S,e) \Leftrightarrow \text{reference}(S, e) \vee (\text{signature-type}(S, e) \wedge \text{non-abstract}(S, e)) \quad \textbf{(8)}$$

$$\text{refer-to}(e) = \{S \mid \text{refer-to}(S, e)\} \quad \textbf{(9)}$$

$$\text{ref-by}(S) = \{e \mid \text{refer-to}(S, e)\} \quad \textbf{(10)}$$

By re-definition of *refer-to*, equation (5) is now also applicable to abstract data types.

4.3. Adding Cohesion to Delta-IC

From a purely relational point of view (i.e., without considering the actual semantics of the entities but only the *refer-to* relationship), one wants to have all subprograms in the cluster to refer to as many variables and types in the cluster as possible to be highly cohesive. Though originally not intended as such, the subtrahend of *Delta-IC* in (5) could be viewed as a way to measure some degree of cohesion along this line: If all subprograms of the cluster access more than one variable in the cluster, the subtrahend is zero. Yet, as already discussed, this kind of cohesion metric is not appropriate, in particular for clusters that contain only one variable or type. An alternative way to define cohesion could be by way of the following equation that yields 1 if all subprograms of the cluster reference all variables and types (let *C* be a cluster, *subs*(*C*) the subprograms of *C*, and *vt*(*C*) the variables and types of *C*):

$$\left(\sum_{e \in \text{vt}(C)} |\text{refer-to}(e) \cap \text{subs}(C)| \right) / (|\text{vt}(C)| \times |\text{subs}(C)|) \quad \textbf{(11)}$$

However, this measurement for cohesion is too strict in practice. Our benchmarks do not contain larger components for which equation (11) would yield a high value. Variables, for example, represent different aspects of a

component and very often only few subprograms of a component deal with all aspects. E.g., a function *size* of a stack would only access the stack pointer but not the stack content.

Again from a purely relational point of view, the entities in the cluster are related to each other because they refer to each other. Note that our definition of *refer-to* yields a bipartite *refer-to* graph, i.e., subprograms are directly related to variables and types but not to other subprograms since the call relationship is not covered by *refer-to*. Likewise, variables and types are neither related. Yet, two subprograms are at least indirectly related if they refer to the same variable or type. To put this on a more formal basis, two entities, E_1 and E_2, are directly related to each other if and only if *refer-to*(E_1, E_2) or *refer-to*(E_2, E_1), denoted by *related*(E_1, E_2). Obviously, *related* is a symmetric relationship. The *related-graph* is immediately induced by the refer-to graph by simply turning the directed *refer-to* edges into undirected *related* edges. Two entities, E_1 and E_2, are said to be *transitively* (or *indirectly*) *related* if and only if there exists a set of entities $\{e_1, e_2, \ldots e_k\}$ such that *related*(e_i,e_{i+1}) for $1 \leq i < k$ and *related*(E_1, e_1) and *related*(e_k, E_2). For example, in the related-graph in Figure 6(b) induced by the refer-to graph in Figure 6(a), f_1 and f_3 are transitively related while f_1 and f_2 are even directly related.

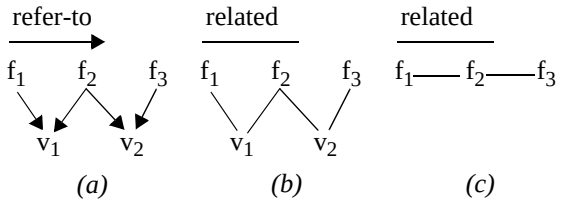

Figure 6: Diverse graph types.

According to the clustering criterion of *Delta-IC*, two entities may only be in the same cluster if they are transitively related to each other, which is intuitive from a purely relational point of view: if two subprograms are not connected via types in their signature or referenced variables or other related subprograms, why should they be considered access functions of the same abstract data type or object? Of course, there are also exceptions, like local utility functions of a component that do not refer to the variables and types in the component but are only called by the other subprograms in the component for a special service. However, such local utility functions are no core access functions of the component and can easily be detected by means of dominance analysis [4, 14].

Now, it could be that the indirect *related* relationship of f_1 and f_3 in Figure 6(b) is only spurious because f_2 is actually a badly designed initialization routine that sets v_1 and v_2. According to the information hiding principle, the system should rather be restructured by providing two initialization routines for the two distinct abstract data objects around v_1 and v_2, respectively, and two calls from f_2 to these initialization routines. In terms of the refer-to graph this would mean to remove the outgoing refer-to edges from f_2 and, consequently, also the related edges in the induced related-graph. The example graph in Figure 6(b) would then be split into two isolated subgraphs and we would consider these subgraphs separate abstract data objects. However, in the general case, removal of one node from the related-graph does not necessarily lead to isolated subgraphs because there could be other entities that hold the graph together. If many entities need to be removed until the graph is finally split into isolated subgraphs, our confidence that these remaining subgraphs are really abstract data objects of their own decreases.

These observations directly correspond to the connectivity concept in graph theory. There are basically two kinds of connectivity measures in graph theory: arc and vertex connectivity. A graph has *vertex connectivity K* if the deletion of any *K*-1 nodes fails to disconnect the graph [7]. Analogously, a graph has *arc connectivity K* if the deletion of any *K*-1 edges fails to disconnect the graph. Algorithms to compute both kinds of connectivity are described in [7].

The upper bound for the arc connectivity of a related-graph is the minimal number of edges of a single node in the graph: If all its edges are removed, the node is isolated from the rest of the graph. In other words, a high arc connectivity requires all subprograms to refer to many variables and types, similar to equation (11). As a matter of fact, if there is a subprogram that accesses only one variable but all other subprograms reference all variables, the arc connectivity measure is even more drastic than equation (11) since the latter would still yield a high value whereas the vertex connectivity of such a graph is only 1. Since we already argued that equation (11) is too strict, vertex connectivity is even less appropriate.

Similarly, a naive application of the vertex connectivity measure to the related-graph is neither appropriate. Since the related-graph is bipartite and subprograms determine the formation of clusters with their patterns of access, we can only consider deleting subprogram nodes for the vertex connectivity when computing the connectivity. This can be achieved by ascertaining the vertex connectivity on the transformed related-graph. The *transformed related-graph* G' of a related-graph G contains only the subprograms of G and there is an edge between two subprograms, S_1 and S_2, in G' if and only if $\exists\ (E \in G)$ *related*(S_1, E) $\wedge$ *related*(S_2, E). For example, the transformed related-graph of the related graph in Figure 6(b) is shown in Figure 6(c).

The vertex connectivity, *VC*, of the transformed related-

graph as a measure for cohesion can be integrated in different ways with the internal connectivity *IC* of *Delta-IC* according to (4) (let *C*(*S*) = *candidate-cluster*(S)):

$$conn_1(\mathrm{S}) = \mathrm{IC(S)} + n \times VC(C(S)) \quad \textbf{(12)}$$

$$conn_2(\mathrm{S}) = \left(\mathrm{IC(S)} + n \times \frac{VC(C(S))}{|\mathrm{subprograms}(C(S)) - 1|}\right) / (n+1) \quad \textbf{(13)}$$

$$conn_3(\mathrm{S}) = \begin{cases} \mathrm{IC(S)} & \text{if } VC(C(S)) \geq m \\ 0 & \text{otherwise} \end{cases} \quad \textbf{(14)}$$

Equation (12) simply adds cohesion to coupling (*n* is used to weigh the influence of cohesion). Since the upper bound of the vertex connectivity is the number of nodes - 1, coupling and cohesion are unbalanced (IC is always in the range of 0 and 1) and should therefore be disregarded. A better alternative is equation (13) which normalizes the vertex connectivity and yields a value between 0 and 1. Parameter *n* can be used to adjust the influence of cohesion versus coupling. In equation (14), the vertex connectivity is used as a filter. All clusters whose cohesion is below a certain threshold *m* will have a connectivity of 0. We tried the latter two alternatives on our benchmarks to obtain quantitative data on the usefulness of these alternative definitions. The next section reports on the result.

5. Benchmark Evaluation

This section provides quantitative results for the following variants of a connectivity measure, *conn*, for the Delta-IC algorithm for reverse engineering in Figure 5:

- **dIC**: conn(S) := ΔIC(S) acc. to (5)
- **IC**: conn(S) := IC(S) acc. to (4)
- **ICVC**: conn(S) := $conn_2$(S) acc. to (13) where *n*=1
- **ICVCF**: conn(S) := $conn_3$(S) acc. to (14) where *m*=2

Note that ICVC for *n*=0 and ICVCF for *m*=0 are equivalent to IC. We measure recall and precision of the automatic techniques by comparing their candidate components to the *reference* components of our benchmark systems (Aero, Bash, CVS, and Mosaic, each at the size of about 35 KLOC and written in C) that were independently and manually detected by software engineers [18]. Recall is — roughly speaking — measured as degree of overlap among matching candidates and references. A detailed definition can be found in [19]. Precision is measured as the number of candidates that do not correspond to a reference component at all (false positives).

The ADO candidates were compared to the ADO and HC references of these benchmarks, whereas the ADT candidates were compared to the ADT and HC references. HCs were also used as references because the techniques that detect ADOs (or ADTs) can also identify HCs at least partially. Because very small and particularly large components are not useful, candidates with less than 3 elements and more than 75 elements were filtered out for the comparison, which reduces the number of false positives. The largest references in the benchmark have about 50 elements such that the candidates could not exceed the references by more than 50%.

The connectivity thresholds of the *Delta-IC* variants were determined by a systematic search for settings that yield the best balance of high recall and low number of false positives for the set of reference components. Table 3 lists the selected thresholds. In practice, one does not have the set of components in advance. Instead, one has to find reference components for a sample of the system either manually or using other techniques. Consequently, the figures below describe the best possible outcome of the *Delta-IC* variants. As a matter of fact, the results are sensitive to the threshold and hence, if a suitable threshold cannot be ascertained, the results will be worse. On the other hand, in an interactive application of this approach, the user would browse the list of candidates ranked according to their connectivity from the highest rank to one that seems doubtful. In other words, an a priori threshold would not be needed.

Recall and false positives are shown in Figure 7 and Figure 8, respectively. The results include the generalization for types. As a comparison point, the charts also contain results for the following other well-known fully automatic connection-based techniques:

- SM: *Same Module* for ADT and ADO recovery [14]
- IA: *Internal Access* for ADT and ADO recovery [32, 13]
- PT: *Part Type* for ADT recovery only [25]
- GR: *Global Reference* for ADO recovery only [32]

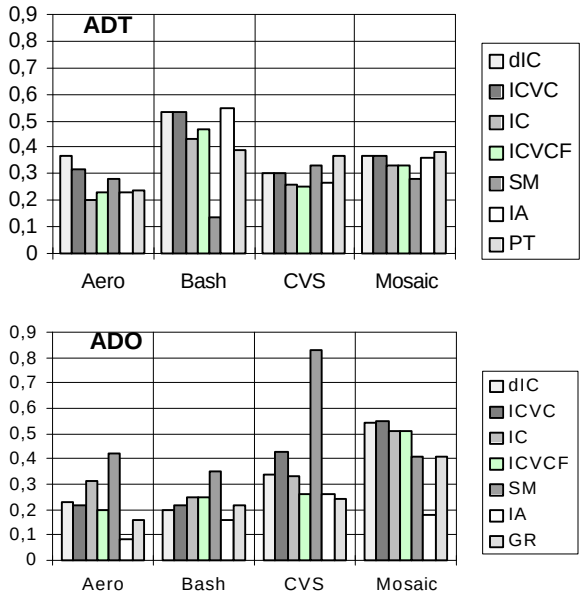

Figure 7: Recall rates.

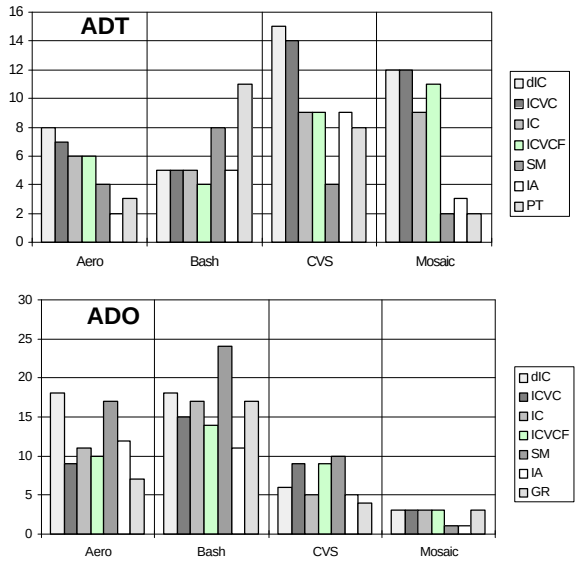

Figure 8: False positives.

Table 3. Thresholds for acceptable connectivity.

System		dIC	ICVC	IC	ICVCF
Aero	adt	-0.24	0.75	0.66	0.60
	ado	0.24	0.80	0.55	0.43
Bash	adt	0.00	0.60	0.0-1.0	0.1-0.8
	ado	0.38	0.91	0.81	0.45
CVS	adt	-0.40	0.76	0.80	0.01
	ado	0.25	0.71	0.65	0.37
Mosaic	adt	-0.01	0.56	0.90	0.00
	ado	-0.40	0.50	0.00	0.00

The quantitative evaluation of the alternative connectivity metrics showed following results:

- in the case of ADO recovery, the *Delta-IC* variants are better than *Internal Access* and *Global Reference*, yet worse than *Same Module* (except for Mosaic)
- in the case of ADT recovery, the *Delta-IC* variants are comparable to other techniques in terms of recall, yet do have a higher number of false positives
- among the *Delta-IC* variants for ADT recovery, dIC and ICVC have a higher recall, but also more false positives; in the case of ADO recovery, IC seems to be the best compromise among the *Delta-IC* variants both in terms of recall and false positives
- the thresholds for acceptable connectivity of all *Delta-IC* variants depend upon the systems; in our evaluation, we obtained the best threshold on the basis of all reference components, hence, the given results are the best ones one would get if one knew the threshold in advance
- as in previous evaluations, even though we actually have got better results with the new *Delta-IC* variants, the results are still not good enough; the best recall rate among all techniques including other well known techniques is only about 40% on average; 83% recall of ADOs for CVS by *Same Module* is an outlier

6. Conclusion

The original *Delta-IC* approach [2] can only detect abstract data objects. In this paper, it was extended to detect abstract data types as well. Furthermore, a variant of *Delta-IC* suitable for reverse engineering was described and alternative combinations of the underlying interconnectivity metric, *IC*, which measures coupling, with a cohesion metric based on vertex connectivity were introduced and evaluated. The quantitative results obtained for ADO recovery are better than two other standard techniques and worse than *Same Module*. However, *Same Module* relies on the assumption that the system is already well decomposed and yields bad results if the related entities are in different modules. For ADT recovery, the *Delta IC* seems to be less suited.

With respect to the classification of automatic component recovery techniques introduced in Section 2, the extension of *Delta-IC* adds a graph-based dimension to the original method, as an analysis of the whole refer-to graph is needed to compute the vertex connectivity used as cohesion metric. The main differences with other graph-based methods are the metric used for *Delta-IC* and the fact that *Delta-IC* considers both coupling and cohesion to generate the clusters, whereas other methods focus on one characteristic. As an example, cohesion is the main driver of *Strongly Connected Component* analysis [4], while *Dominance* analysis focuses on coupling [4].

Future work needs to address the effect on detection quality when connectivity thresholds are calibrated on smaller samples of the system. In an evaluation of another metric-based approach, namely, *Similarity Clustering*, we could show that a sample of 20% of the components of the system seems to be enough [18]. Whether this is also true for *Delta-IC*, has to be shown. Furthermore, we want to investigate the effect of the tolerance parameter *p* of the partial subset relationship (7) on overlap among candidates as well as detection quality. Another interesting avenue will be to examine whether articulation points in the related-graph are good candidates for slicing. A long term goal is to explore whether data flow analyses can improve the insufficient effectiveness of connection-based techniques.

References

[1] Belady, L.A., Evangelisti, C.J., 'System Partitioning and its Measure', Journal of Systems and Software, 2(1), pp. 23-29, February 1982.

[2] Canfora, G., Cimitile, A., Munro, M., 'An Improved Algorithm for Identifying Objects in Code', Journal of Software Practice and Experience, 26(1), pp. 25–48, January 1996.

[3] Canfora, G., Cimitile, A., De Lucia, A., and Di Lucca, G. A. 'A Case Study of Applying an Eclectic Approach to Identify Objects in Code', Proc. of 7th Workshop on Program Comprehension, Pittsburgh, PA, May 1999, IEEE Comp. Soc. Press, pp. 136-143.

[4] Cimitile, A. and Visaggio, G., 'Software Salvaging and the Call Dominance Tree', Journal of Systems Software, 28:117–127, 1995.

[5] Choi, S.C., Scacchi, W., 'Extracting and Restructuring the Design of Large Systems', IEEE Software, 7(1), pp. 66-71, January 1990.

[6] Doval, D., Mancoridis, S., Mitchel, B.S, Chen, Y., Gansner, E.R., 'Automatic Clustering of Software Systems using a Genetic Algorithm', Proceedings of the International Conference on Software Tools and Engineering Practice, August 1999.

[7] Even, S., *Graph Algorithms*, Pitman Publishing Ltd., 1979.

[8] Gall, H., Klösch, R., 'Finding Objects in Procedural Programs: An Alternative Approach', Proceedings of the Second Working Conference on Reverse Engineering, pp. 208-216, Toronto, July 1995.

[9] Gall, H., Klösch, R., and Weidl, J., 'Resolving Uncertainties in object oriented re-architecturing of procedural code', 7th International Conference on Information Processing and Management of Uncertainty in Knowledge Based Systems, July, 1998.

[10] Ghezzi, G., Jazayeri, M., Madrioli, D., 'Fundamental Software Engineering', Prentice Hall International, 1991.

[11] Girard, J.F and Briand, L., 'Reengineering Concepts, Techniques and Tools for Component Extraction', technical report CRIM95/04-26, May, 1996.

[12] Girard, J.F., Koschke, R., Schied, G., 'A Metric-based Approach to Detect Abstract Data Types and Abstract State Encapsulation', Journal on Automated Software Engineering, 6, pp. 357-386, Kluwer 1999.

[13] Girard, J.F., Koschke, R., 'A Comparison of Abstract Data Type and Objects Recovery Techniques', Journal Science of Computer Programming, Volume 36, Issue 2-3, Elsevier, March 2000.

[14] Girard, J.F., Koschke, R., 'Finding Components in a Hierarchy of Modules: a Step Towards Architectural Understanding', International Conference on Software Maintenance, pp. 58-65, Bari, October 1997.

[15] Hutchens, D.H., Basili, V.R., 'System Structure Analysis: Clustering with Data Bindings', IEEE Transactions on Software Engineering, SE-11(8), pp. 749-757, August 1985.

[16] Kazman, R., Carrière, S.J., 'Playing detective: reconstructing software architecture from available evidence', Technical Report CMU/SEI-97-TR-010, ESC-TR-97-010, Software Engineering Institute, Pittsburgh, USA, 1997.

[17] Koschke, R., 'A Semi-Automatic Method for Component Recovery', Proceedings of the Sixth Working Conference on Reverse Engineering, pp.256-267, Atlanta, October 1999.

[18] Koschke, R., 'Atomic Architectural Component Detection for Program Understanding and System Evolution', Ph.D. thesis. University of Stuttgart, 2000.

[19] Koschke, R. and Eisenbarth, T. 'A Framework for Experimental Evaluation of Clustering Techniques', Proc. of the International Workshop on Program Comprehension, IEEE Computer Society Press, June, 2000.

[20] Lakhotia, A., 'A Unified Framework for Expressing Software Subsystems Classification Techniques', Journal Systems Software, Elsevier Science Publisher, 36, pp. 211-231, 1997

[21] Lindig, C., Snelting, G., 'Assessing Modular Structure of Legacy Code Based on Mathematical Concept Analysis', Proceedings of the Nineteenth International Conference on Software Engineering, Boston, 1997.

[22] Macro, A. and Buxton, J., *The Craft of Software Engineering*, Addison-Wesley, Reading, MA, 1987.

[23] Müller, H., Wong, K., Tilley, S., 'A Reverse Engineering Environment Based on Spatial and Visual Software Interconnection Models', Proceedings of the Fifth ACM SIGSOFT Symposium on Software Development Environments, pp 88-98, Tyson's Corner, December 1992.

[24] Müller, H.A., Orgun, M.A., Tilley, S.R., Uhl, J.S., 'A Reverse Engineering Approach to Subsystem Structure Identification'. Journal of Software Maintenance: Research and Practice, 5(4), pp. 181-204, December 1993.

[25] Ogando, R.M., Yau, S.S., Wilde, N., 'An Object Finder for Program Structure Understanding in Software Maintenance', Journal of Software Maintenance, 6(5), pp. 261–83, September-October 1994.

[26] Patel, S., Chu, W., Baxter, R., 'A Measure for Composite Module Cohesion', Proceedings of the Fourteenth International Conference on Software Engineering, pp. 38-48, Melbourne, May 1992.

[27] Sahraoui, H., Melo, W, Lounis, H., Dumont, F., 'Applying Concept Formation Methods to Object Identfication in Procedural Code', Proceedings of the Twelfth Conference on Automated Software Engineering, pp. 210-218, Nevada, November 1997.

[28] Schwanke, R. W., 'An Intelligent Tool for Re-engineering Software Modularity', Proceedings of the International Conference on Software Engineering, pp. 83–92, May 1991.

[29] Siff, M., Reps, T., 'Identifying Modules via Concept Analysis', Proceedings of the International Conference on Software Maintenance, pp. 170-179, Bari, October 1997.

[30] Valasareddi, R.R., Carver, D.L., 'A Graph-based Object Identification Process for Procedural Programs', Proceedings of the Fifth Working Conference on Reverse Engineering, pp. 50-58, Honolulu, October 1998.

[31] Weiser, M., 'Program Slicing', IEEE Transactions on Software Engineering, vol. SE-10, no. 4, July, 1984.

[32] Yeh, A.S., Harris, D., Reubenstein, H., 'Recovering Abstract Data Types and Object Instances From a Conventional Procedural Language', Proceedings of the Second Working Conference on Reverse Engineering, pp. 227–236, July 1995.

[33] Yourdon, E. and Constantine, L.L., *Structured Design*, Prentice Hall, Englewoood Cliffs, NJ, 1979.