A Metric-based Approach to Detect Abstract Data Types and State Encapsulations

Jean-François Girard

Fraunhofer Institute for
Experimental Software Engineering
Sauerwiesen 6
D-67661 Kaiserslautern, Germany
girard@iese.fhg.de

Rainer Koschke, Georg Schied

University of Stuttgart
Breitwiesenstr. 20-22
D-70565 Stuttgart, Germany
{koschke, schied}@informatik.uni-stuttgart.de

Abstract

*This article presents an approach to identify abstract data types (**ADT**) and abstract state encapsulations (**ASE**, also called abstract objects) in source code. This approach groups together functions, types, and variables into ADT and ASE candidates according to the proportion of features they share. The set of features considered includes the context of these elements, the relationships to their environment, and informal information.*

A prototype tool has been implemented to support this approach. It has been applied to three C systems (each between 30-38 Kloc). The ADTs and ASEs identified by the approach are compared to those identified by software engineers who did not know the proposed approach. In a case study, this approach has been shown to identify, in most cases, more ADTs and ASEs than five published techniques applied on the same systems. This is important when trying to identify as many ADTs and ASEs as possible.

1. Introduction

The Bauhaus project[1] aims at recovering the software architecture of a system "as it is implemented". That is, recovering multiple views which would indicate the main **components** of the system, their **connectors** (how they communicate), and the **constraints** on these connectors and components.

This article presents an approach which can be used as a first step in components identification. A case study [Gira97a] suggests that abstract data types and abstract state encapsulations are two instances of the smallest components which are significant at the architectural level. The authors have named such building blocks **atomic components**, because they can be used to build larger components.

According to Sommerville [Somm92] an **abstract data type** (**ADT**) is an abstraction of a type which encapsulates all the type's valid operations and hides the details of the implementation of those operations by providing access to the types exclusively through a well-defined set of operations.

An **abstract state encapsulation** (**ASE**) is a group of global variables together with the routines which access them. These clusters are also called abstract objects [Ghez91] or object instances [Yeh95].

Cross-breedings of ADTs and ASEs are called **hybrid components**. They consist of routines, variables, and types.

These three types of atomic components are important to the authors' research on architecture recovery, but they are also of general importance. Their role has been recognized to provide information hiding and thus to support maintainability [Ghez91] and reuse [Somm92].

However, these atomic components are not always explicitly captured in the source code, because programming languages like C do not provide good support to express them or software development does not always exploit them. Consequently, reverse engineering techniques are required to identify them in these circumstances.

This article presents a reverse engineering approach, called **similarity clustering**, which groups together functions, types, and variables into ADT or ASE candidates according to a similarity metric. The set of features considered by this metric includes the context of these elements, the relationships between these elements, and

1. Research collaboration between the Fraunhofer IESE and the University of Stuttgart.

informal information. In order to evaluate the proposed similarity clustering approach, it was applied to three C systems (between 30-38 Kloc) and the results were compared to the atomic components identified by software engineers. Five other published atomic component identification techniques were applied to these systems. On this benchmark, similarity clustering approach identifies, in most cases, more correct ASEs and ADTs than the other techniques. This is important in order to recover as many ADTs and ASEs as possible.

Paper Overview

The remainder of the article is organized as follows: Section 2 introduces the system graph abstraction used to define the similarity metric at the core of the proposed approach. Section 3 presents the approach. Section 4 presents related research. Section 5 describes the experiment setup used to evaluate atomic components identified by automatic techniques. Section 6 discusses results and summarizes the results of a case study [Gira97b] comparing some of them to the proposed approach. Section 7 concludes and proposes further research.

2. A Conceptual Model for Programming Language Entities

The ***resource flow graph*** is a graph abstraction of a system which is used to define the similarity metric at the core of similarity clustering. It captures typical relationships between routines, variables, and user-defined types, the building blocks of atomic components. The relationships considered in this approach are illustrated in the entity-relationship diagram of Figure 1.

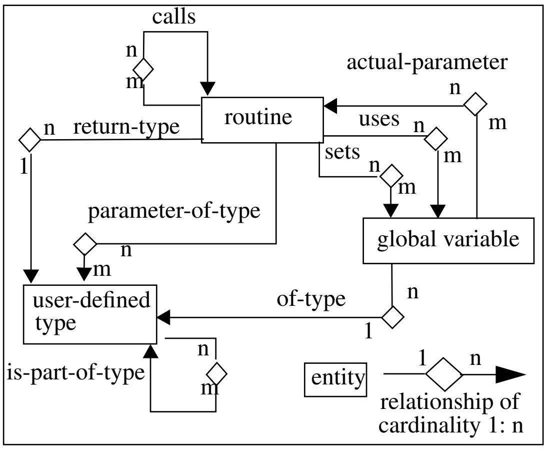

Figure 1. An entity-relationship model for elements of atomic components.

This entity-relationship model defines the structure of a resource flow graph which is an abstraction of the source code. Nodes of this graph stand for the entities and edges express the relationships among entities. In the following section, the resource flow graph is used to define the similarity metrics.

3. Similarity Metric Approach

This section presents the similarity clustering approach. It first gives an overview of the approach. Then it discusses the various aspects of similarity which are combined in the approach.

This approach is inspired by Schwanke's work [Schw91] aimed at detecting subsystems using a similarity metric between routines. The clustering similarity approach applies the idea of this work to atomic components identification by generalizing this metric, adding informal information, edge-dependent weights, and adapting many of its parameters. The two approaches will be contrasted in more detail in section 4.

3.1. Approach Overview

The similarity clustering approach groups entities (functions, user-defined types, and global variables) according to the proportion of features (entities they access, their name, the file where they are defined, etc.) they have in common. The intuition is that if these features reflect the correct direct and indirect relationships between these entities, then entities which have the most similar relationships should belong to the same ADT or ASE.

Functions, variables, and types are grouped according to the following algorithm:

Place each entity in a group by itself
repeat
Identify the most similar groups
Combine them
until the existing groups are satisfactory
remove groups which are not valid atomic components

In each iteration of this algorithm, a similarity metric measures the proportion of features which are shared. The algorithm terminates when "existing groups are satisfactory". In the experiment reported in this paper, groups are considered satisfactory when the most similar groups have a similarity which is below a certain threshold. This threshold is established experimentally on systems where atomic components are known.

The similarity metric is constructed of three layers:

- The similarity between **two groups of entities** which is defined in terms of similarity between entities across groups.

- The similarity between **two entities** which is a weighted sum of various aspects of similarity.
- Each specific **aspect of similarity** between two entities.

The similarity between groups of entities used in this paper is defined as the average between all pairs of similarities in the two sets:

$$Sim(S_1, S_2) = \frac{\sum_{(s_i \in S_1, s_j \in S_2)} Sim(s_i, s_j)}{size(S_1) \times size(S_2)}$$

The similarity between two entities is the weighted sum of various aspects of similarity. The following **aspects** are included in the similarity metric for the experiment reported in this paper:

- Direct relations
- Indirect relations
- Informal information

Direct relations are relations between the two entities compared. Indirect relations are relations with common third entities. Informal information is the information in a program source code which is not captured by the semantics of programming languages, but is used by programmers to communicate the intent of a program (e.g., comments, identifier names, file organization, etc.)

The similarity between two entities A and B is defined as follows (the factors x_i are used to adjust the influence of the diverse specific similarities):

$$Sim(A, B) = [x_1 \cdot Sim_{indirect}(A, B) + x_2 \cdot Sim_{direct}(A, B) + x_3 \cdot Sim_{informal}(A, B)] / (x_1 + x_2 + x_2)$$

Each aspect is normalized to obtain values between 0 and 1, so the resulting similarity is also normalized.

3.2. Direct Relations

Direct relations represent immediate connections between two entities and are generally more important than indirect relations and informal information. Their contribution to the similarity is computed as $Sim_{direct}(A, B)$ which yields a value between 0 and 1. In terms of the resource flow graph, this is defined as the weighed sum of edges between A and B divided by the weighed sum of all possible edge types between A and B:

$$Sim_{direct}(A, B) = \frac{W(link(A, B))}{W(\text{all-links}(nodetype(A), nodetype(B)))}$$

where *link (A, B)* denotes the actual links between A and B, *all-links()* denotes all edge-types which are possible in a resource flow graph between two given node types. $W(X) = \sum_{x \in X} weight(x)$ where $weight(x) \geq 0$ is a weighting factor which allows to give certain features more influence on the global value of the metric.

Weights

The weight factors are introduced to give more influence to certain features. The weights are the first factors used to improve the precision of the approach.

Shannon Information Content

One weighting factor *weight(x)* used in this paper is Shannon's information content [Shan72] from information theory:

$$weight(x) = -\log(probability(x))$$

where *probability(x)* is replaced by the fraction of all entities that have x in common. The hypothesis is that rarely used entities are more significant than frequent entities. For example, an error variable which is used everywhere in the system is less distinctive than a variable that is only used by a small portion of the system.

Relation Type

An alternative to Shannon's information content is to assign fixed edge-weights to the kind of edges between entities. For example, when looking for an ADT, edges connected to user-defined types are more important than call edges, hence should be given more weight.

These fixed weights are used in some experiments. In other experiments they were combined with Shannon's information content by multiplying the two values.

3.3. Indirect Relations

Indirect relations capture the proportion of common features two entities share. In terms of the resource flow graph, these features are neighbors of the entities compared. The similarity aspect for indirect relations uses the following formula:

$$Sim_{indirect}(A, B) = \frac{Common(A, B)}{Common(A, B) + d \cdot Distinct(A, B)}$$

where *Common (A,B)* reflects the number of common features of A and B, *Distinct (A,B)* reflects the number of distinct features and $d \geq 0$ is a parameter which regulates the importance given to distinct features. In terms of the resource flow graph, they are defined as follows:

$$Common(A, B) = W(\textit{edges-with}(A) \cap \textit{edges-with}(B))$$

$$Distinct(A, B) = W(\textit{edges-with}(A) \oplus \textit{edges-with}(B))$$

The operator $\oplus$ denotes the symmetric difference for sets. The term *edges-with(B)* returns a set of nodes which are connected to B. The exact set of nodes depends on whether the direction of edges is considered or not.

3.4. Informal Information

Programmers capture part of the meaning of programs in comments and in the name of functions, variables, and types. This helps them and other programmers find their way around in a program. Another guide in a program is the file organization: related functions, variables, and types are often put together in one file. Both of these means of communication among programmers are examples of informal information. Usually, informal information is ignored by reverse engineering techniques (a notable exception is [Bigg89]) which focus on the information kept by a compiler.

This section discusses how the information contained in the names of program identifiers and file organization can be relevant to the identification of atomic components. It also describes how they are taken into account by the proposed approach.

3.4.1. Names of Identifiers

The naming of functions, variables, and types is an important source of information about a program given to a human reader. It has been observed [Bigg89] that even the author of a program has difficulties in recognizing the purpose of an excerpt from his code once significant identifier names have been replaced by insignificant ones. (e.g., top_stack -> f1). The naming of identifiers also convey information relevant to the identification of atomic components. For example, in one of the systems investigated routines that belong to an abstract data type list had similar names: list_insert, list_remove, and list_create.

Two naming conventions are widely used for long identifiers built from many words: separate words with underscore ('_') or start each new word with a capital letter (e.g., InsertWord). The following metric based on the number of common words between two identifiers exploits these conventions:

$$Sim_{words}(X, Y) = \frac{|\mathrm{words}(X) \cap \mathrm{words}(Y)|}{|\mathrm{words}(X) \cup \mathrm{words}(Y)|}$$

When these conventions are not used, identifiers are frequently constructed using a common prefix or postfix. In this case the following metric is used:

$$Sim_{suffix}(X, Y) = \begin{cases} 1 & X = Y \\ \dfrac{\mathrm{prefix}(X, Y) + \mathrm{postfix}(X, Y)}{1 + \mathrm{prefix}(X, Y) + \mathrm{postfix}(X, Y)} & X \neq Y \end{cases}$$

where prefix and postfix are the lengths of the common pre- and postfix of their two arguments, if the length is longer than three characters; otherwise it is zero.

3.4.2. Organization of Files

The division of a program into files also conveys some information about the meaning of a program. Related functions and variables are often put in the same file or in files with a common substring in their name (e.g., client-db and client-service). The previous metric for identifier name similarity based on pre- and postfixes is used to compare file names without extensions (i.e., only file in file.c or file.h)

$$Sim_{filename}(X, Y) = Sim_{suffix}(fname(X), fname(Y))$$

4. Related Research

This section presents a brief overview of the related research in reverse engineering and similarity metrics.

4.1. Detection of ADTs and ASEs or Objects

There is a rich literature on the detection of ADTs and ASEs or object instances from systems written in procedural languages (usually C). Many techniques [Ogan94, Gira97a, Yeh95, Canf93a, Canf94] that aim at ADT recovery focus on the relations between types and the parameters and the return type of a function (called function signature). These relations are usually represented by edges in a graph where the nodes correspond to functions and user-defined data types of a system. The basic idea of these techniques is that the set of connected components of this graph form the set of ADT candidates. However, these candidates are often too large, because of functions which examine or modify structures of many types. As a result, these ADT candidates often combine and thereby hide multiple correct reference ADTs.

Each of the following techniques proposes a different heuristic to avoid these large candidates by imposing additional conditions on the connected components:

- *Same Module heuristic* [Gira97a]: It groups only those routines, global variables, and types together that are declared in the same file.
- *Part Type heuristic* [Ogan94]. It filters out types in a parameter list that are part of another type in the list.
- *Internal Access heuristic* [Yeh95]: It filters out routines which do not access internal parts of record types that are part of an ADT.
- *Enumeration & Dominance heuristic* [Canf94]: It filters out enumerations and subranges from the types considered to form an ADT when the ADT is too large. It also uses the dominance relation on the call graph to identify functions which are called only from an ADT in order to package them with the ADT.

Our similarity clustering considers the same relations between functions and types, but it also takes into account the relations with the context of entities. Like the Same Module heuristic, similarity clustering considers the file where types and functions are defined, but the influence of the file organization on the metric can be lowered when it is not a important factor. Like Enumeration and Dominance heuristic, similarity clustering considers the call relationship between routines.

Yeh et al. [Yeh95] and Canfora et al. [Canf93] identify ASEs or object instances based on the set and use relationship between functions and global variables. These relations are usually represented by edges in a graph where the nodes correspond to functions and global variables. These approaches can be summarized as follows:

- *Delta IC* (an adaptation of [Canf93b]) It generates potential ASEs using a fixed pattern, then evaluates the improvement on a connectivity metric due to the introduction of a potential candidate. If the improvement is good enough, then the candidate is kept, otherwise it is rejected.
- *Object Instance heuristic* [Yeh95]: It computes the connected components of this graph and considers them as ASE candidates.

Our similarity clustering differs from these techniques in that it takes into account how frequently a variable is used while creating candidates. This addresses one of the two causes of massive components reported by Yeh et al. [Yeh95] of "external variables or structures used throughout the system". Giving more weight to variables used infrequently can alleviate this problem.

4.2. Schwanke's Approach

In [Schw91], Schwanke defines a similarity metric used to group functions into modules. This approach differs from previous approaches in that it generates candidates using a metric which provides a more refined view on the relationships between program entities. The previous approaches compute connected components or other fixed patterns in a graph. In contrast, whether these patterns are present or not, Schwanke's approach deals with continuous values which reflect the degrees of similarity. This is computed as $\mathrm{Sim}_{\mathrm{indirect}}$ given in section 3.3.

The similarity clustering approach exploits these nuances, but extends the metric in the directions summarized in Table 1.

4.3. Similarity

Similarity metrics have been researched extensively by the AI community, in particular the case-based reasoning community. Many aspects of this research focus on specific needs of this community, for example Aamodt [Aamo94] records knowledge about the field in order to provide explanations. However, some references like Richter [Rich92] present a good overview of the basic ideas and introduce learning approaches which can be applied to discover the appropriate similarity. A systematic search for the best possible similarity metric is beyond the objective of this research. However, it is future work to identify what could be applied from related work on similarity metrics.

Table 1: Comparison with Schwanke's Approach

	Schwanke's Approach	Article's Approach
Domain	• routines	• routines • user-defined types • global variables
Weights	• Shannon info.	• Shannon info. • edges weight which reflect the type of relation between entities
features	• usage of non-local names	• type relations • usage relations • call relations
informal info.	• not used	• tokens in identifiers • pre- and postfix in identifiers • file
direct links	• values = 0 or 1	• continuous values between 0 and 1
similarity between groups	• maximum similarity between elements	• average similarity between elements

5. Experiment Setup

In order to evaluate our similarity clustering approach, it was applied to three medium-size programs, and the results were then compared to the atomic components identified by software engineers. The components identified by the approach are called the *candidate components*. The atomic components identified by software engineers will be called *reference components*. This section summarizes the experimental setup and the analysis method used.

5.1. Systems Studied

The analyses described above were applied to three medium size C programs (see Table 2 for their characteristics). Aero is an X window system-based simulator for rigid body systems [Kell95], bash is a Unix shell, and CVS is a tool for controlling concurrent software development.

Table 2. Systems Studied

System Name	Version	Lines of Code	# User Defined Types	# Global Variables	# User Defined Routines
aero	1.7	31 Kloc	57	480	488
bash	1.14.4	38 Kloc	60	487	1002
cvs	1.8	30 Kloc	41	386	575

5.2. Human Analysts

Five software engineers were given the task of identifying atomic components in each system. These systems were unknown to them. There was no overlap of their work. They needed about 20 hours for each system to gather the atomic components of the respective systems.

Details concerning the guidelines given to the software engineers, their experience,and how the task was divided among them are described in [Gira97b]. Table 3 shows the respective numbers of all forms of atomic components (abstract data types, abstract state encapsulation, hybrid atomic components) that were identified by the group of software engineers for each studied system.

Table 3. Reference Atomic Components.

System	#ADT	#ASE	#Hybrid
Aero	10	16	1
Bash	22	15	6
CVS	13	36	6

The variation in experience and the number of people working on each system prevent comparison of different techniques across systems until cross-validation and future studies characterize the importance of these factors. However, preliminary inspection of the identified atomic components found no aberrant components. Furthermore, the fact that none of the software engineers knew the automatic analyses to be applied to the systems, prevented bias toward a specific technique.

5.3. Comparison of Candidate and Reference Components

This subsection explains how imperfect matches between candidate and reference components are compared and classified.

Candidate components Cs and reference components Rs are compared using an approximate matching to accommodate the fact that the distribution of functions, global variables, and types into atomic components is sometimes subjective. We treat one component S as part of another component T (denoted by S << T) if at least 70 percent of the elements of S are also in T.

Based on this approximation, the generated candidates are classified into 3 categories according to their usefulness to a software engineer looking for atomic components:

- **Good** when the match between a candidate C and a reference R is close (i.e., C << R and R << C).

  Matches of this type require a quick verification in order to identify the few elements which should be removed or added to the atomic component.
- **Ok** when the relationship holds only in one direction for candidates C_i and references R_i:
  - C_i << R, but not R << C_i (i > 0)
  - R_i << C, but not C <<R_i (i > 0)

  Partial matches of this type require more attention to split, combine, or refine a component.
- **Bad** candidate components are not close enough to the reference components to guide the software engineer's work.

5.4. Accuracy

In order to indicate the quality of imperfect matches of candidate and reference components, an accuracy factor has been associated with each match. The accuracy between a candidate C and a reference R is computed by the following formula:

$$accuracy(C, R) = \frac{|C \cap R|}{|C \cup R|}$$

For matches between more than two components (n~1 and 1~n) the union of all elements of the n components is used to compute the accuracy. The accuracy is not defined for n~m matches, because the m references are not always unique.

6. Results

This section compares the atomic components recovered by similarity clustering with those recovered by other techniques. Then it discusses the impact of false positives.

Benchmark Results.

The case study [Gira97b] applied the following techniques to recover atomic components on the three systems described here and used the same evaluation setup:

- same module [Gira97a]
- part type [Ogan94]
- internal access (an adapted version of [Yeh95] that combines results of Yeh et al.'s internal access and object instance heuristic).
- delta IC (an adaptation of [Canf93b])

- Schwanke (a minimal adaptation of [Schw91]) which is modified only to consider variables and types as well as functions. All other parameters remain as described in [Schw91].

The number of recovered ADTs and ASEs for each system using the above techniques are reported in the case study [Gira97b]. There, the results are analyzed in more detail. In Table 4 we contrast these results with the outcome of our new analyzing technique (last line of the table). The table entries correspond to the number of components (good + ok) recovered by each technique multiplied by their accuracy. It can be observed that similarity clustering identifies considerably more ADTs and ASEs than the other techniques except for the ASEs of CVS and Aero. For approaches that are not able to detect a certain kind of atomic component, the respective entries in Table 4 are left empty. Because of lack of space we leave out the results for hybrid components. They confirm the results for ADT and ASE detection.

Table 4: Recovery Summary

Technique	aero		bash		cvs	
	adt	ase	adt	ase	adt	ase
Same Module	0.3	8.58	0.78	6.9	6.21	26.81
Part Type	0.52	----	7.14	----	6.39	----
Internal Access	1.40	2.90	0.1	2.20	4.8	3.72
Delta IC	----	2.82	----	4.97	----	9.12
Schwanke	0	1.40	0.60	2.3	2.88	7.48
Similarity Clustering	3.91	7.64	10.78	6.8	10.58	25.35

However, identifying these additional components does come with a high price: with the current implementation, the number of false positives (bad category) is significantly larger as Table 5 shows.

Table 5: False Positive Atomic Components

Technique	aero	bash	cvs
Same Module	1	7	1
Internal Access	6	54	21
Part Type	2	15	9
Delta IC	9	43	0
Schwanke	40	121	46
Similarity clustering	69	209	45

In gathering the results presented here, we used 0.15 (aero & cvs) and 0.25 (bash) as thresholds for the clustering algorithm outlined in section 3.1. In practice, when the number of candidates is too large, one can decide to modify the threshold to obtain less candidates or to use the other methods. For example, when the threshold for bash is raised from 0.3 to 0.4, the number of candidates drops from 209 to 135 candidates, but the number of good matches drops from 6 to 2. It is a topic of further research to find a way to reduce the number of false positives.

7. Conclusions

In this article, we presented a new approach to extract **atomic components** (abstract data types and abstract state encapsulations in this paper) from source code. At the core of our work is a clustering method based on a similarity metric between elementary program entities (e.g., routines, types, and variables) inspired by the work of Schwanke [Schw91]. In contrast to the original approach of Schwanke, we generalized the similarity metric to take into account edge-dependent weights and informal information extracted from identifier and file names.

Our experimental results show that these extensions improve the quality and quantity of atomic components found. We evaluated the performance of this approach in the context where it is important to recover as many of these as possible. We compared the components identified by a group of software engineers on three C systems (30-38 Kloc) to the components identified by similarity clustering and five other published approaches. In most cases, our similarity clustering approach identifies more atomic components than the other approaches.

However, the goal of recognizing atomic components of the architecture of software systems is only partially solved. Though giving better results than other methods proposed in the literature, our clustering method recovers only a portion of them and would benefit from being combined with the results of other approaches. Also, the price to pay for these results is a high number of false positives.

In practice, this approach would be applied to a system to suggest a list of candidates which are to be reviewed by software engineers. Reviewing candidates is much faster than looking for the atomic components in the source code. Finding atomic candidates in the code took on average 20 hours for each of the systems studied.

When there are too many candidates to review for the time available, the practitioner can adjust the threshold to reduce the number of candidates or he might select to use an alternative approach. In both cases, the case study reported in this paper could help him amke an informed decision, being aware of the trade-offs involved.

In summary, despite its drawbacks, the similarity clustering approach is a useful tool for identifying atomic components. It is a good complement to existing approaches and can be combined with those.

Future work

- Develop means for reducing the number of false positives in candidates. Possibilities investigated include statistical filtering of candidates and better parametrization.
- Study the relations between the various parameters of the approach trying to find out which ones are system-dependent.
- Develop a systematic process to establish the parameters and report experimental results.
- Explore how the various approaches for identifying atomic components can be combined.

Acknowledgment

We would like to thank Jean-Marc DeBaud and Erhard Plödereder for their support to this project and their insightful inputs to this research. We would like to express special thanks to Joachim Bayer, Hiltrud Betz, Kristina Lundqvist, Minna Mäkäräinen, and Klaus Schmid for their manual analysis of the three subject systems used in our comparison.

References

[Aamo94] A. Aamodt. *Topic in Case-Based Reasoning*, chapter Explanation-Based Case-Based Reasoning, pages 274–288. Springer Verlag, 1994.

[Bigg89] T. J. Biggerstaff. Design recovery for maintenance and reuse. *IEEE Computer*, 22:36–49, July 1989.

[Canf93a] G. Canfora, A. Cimitile, and M. Munro. A reverse engineering method for identifying reusable abstract data type. In *Working Conference on Reverse Engineering*, pages 73–82. May 1993.

[Canf93b] G. Canfora, A. Cimitile, M. Munro, and C. J. Taylor. Extracting abstract data type from C programs: A case study. In *International Conference on Software Maintenance*, pages 200–9. September 1993.

[Canf94] G. Canfora, A. Cimitile, M.Tortorella, and M. Munro. A precise method for identifying reusable abstract data types in code. In *International Conference on Software Maintenance*, pages 404–413. September 1994.

[Ghez91] C. Ghezzi, M. Jazayeri, and D. Madrioli. *Fundamental Software Engineering*. Prentice Hall International, 1991.

[Gira97a] J.F Girard and R. Koschke. Finding components in a hierarchy of modules: a step towards architectural understanding. In *International Conference on Software Maintenance*, 1997.

[Gira97b] J.F Girard, R. Koschke, and G. Schied. Comparison of abstract data type and abstract state encapsulation detection techniques for architectural understanding. In *Fourth Working Conference on Reverse Engineering*, October 1997.

[Kell95] H. Keller, H. Stolz, A. Ziegler, and T. Bräunl. Virtual mechanics simulation and animation of rigid body systems with aero. *Simulation for Understanding*, 65(1):74–79, July 1995.

[Ogan94] R.M. Ogando, S.S. Yau, and N. Wilde. An object finder for program structure understanding in software maintenance. *Journal of Software Maintenance*, 6(5):261–83, September-October 1994.

[Rich92] Richter. Classification and learning of similarity measures. In *Annual Conference of the German Society for Classification*, number 16th. Springer Verlag, 1992.

[Schw91] R. W. Schwanke. An intelligent tool for re-engineering software modularity. In *International Conference on Software Engineering*, pages 83–92, May 1991.

[Shan72] C. E. Shannon. *The mathematical theory of communication*. Urbana: Univ. of Ill. Press, 1972. ISBN 0-252-72548-4.

[Somm92] I. Sommerville. *Software Engineering*. Addison Wesley, fourth edition, 1992.

[Yeh95] A.S. Yeh, D.Harris, and H. Reubenstein. Recovering abstract data types and object instances from a conventional procedural language. In *Second Working Conference on Reverse Engineering*, pages 227–236, Los Alamitos, California, July 1995.