Automated Software Engineering 6, 357–386 (1999)

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

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

RAINER KOSCHKE koschke@informatik.uni-stuttgart.de
GEORG SCHIED schied@informatik.uni-stuttgart.de
University of Stuttgart, Breitwiesenstr. 20-22, D-70565 Stuttgart, Germany

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, named similarity clustering, 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 or other automatic approaches. Within this case study, this approach has been shown to have a higher detection quality and to identify, in most of the cases, more ADTs and ASEs than the other techniques. In all other cases its detection quality is second best. N.B. This article reports on work in progress on this approach which has evolved since it was presented in the original ASE97 conference paper.

Keywords: reverse engineering, (re-)modularization, abstract data type and object identification, clustering, technique comparison, case study

1. Introduction

1.1. Context

Providing an explicit architectural description (Garlan and Shaw, 1993; Perry and Wolf, 1992) of a software system has been proposed as a way to offer an effective basis for reuse, to reduce the erosion of the overall system structure, and to support dependency and consistency analysis (Perry and Wolf, 1992). Currently, most of the software architecture community focuses on defining and experimenting with formalisms to capture architecture while the system is specified and developed (Dean and Cordy, 1995; Shaw et al., 1995; Luckham et al., 1995). However, there is a large body of existing code which needs to be maintained and would also benefit from an architectural description. Thus, there is a need to *recover architectural descriptions for existing systems.*

The Bauhaus project, a research collaboration between the Fraunhofer Institute for Experimental Software Engineering and the University of Stuttgart, aims at recovering the software architecture of a system as multiple views which describe the main *components* of the system, their *connectors* (how they communicate), and the *configurations* (protocols, dependencies, and overall organization) of these connectors and components.

1.2. A step toward component recovery

Constructing the main components of an architectural description of a system directly from its functions, types, and global variables would require bridging a large conceptual gap. A more conservative alternative is to proceed by successive abstraction. This is the path selected by the authors. This article presents a technique to identify instances of the following three kinds of components:

- An *abstract data type (ADT)* (Liskov and Zilles, 1974) 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 instances of such a type 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 (Ghezzi et al., 1991) or object instances (Yeh et al., 1995).
- Cross-breedings of ADTs and ASEs are called *hybrid components (HC)*. They consist of routines, variables, and types.

The main difference between an ADT and an ASE is that one can have any number of instances of an ADT by declaring objects of this type whereas there is always exactly one ASE. We have named these components *atomic components* because they are among the smallest that are significant at the architectural level (a case study by Girard and Koschke (1997) suggested this conclusion).

1.3. Usefulness of atomic component recovery

These atomic components provide a good starting low-level abstraction according to our initial experience. When software engineers were asked to identify atomic components in a set of C systems without the help of automatic techniques, they obtained ADTs, ASEs, and HCs to which they were able to assign significant names and concepts in most cases. During a later case study, one software engineer who had to describe the architecture of an existing system after identifying its atomic components, reported that the atomic components constitute good background information to support this task. In fact, larger components often contain atomic components. So identifying them can help in recognizing the larger components. For example, in Bash, a system used in our experiment, the command handler is decomposed into many subcommand handlers (if_com, for_com, select_com, while_com, group_com, simple_com, . . .), which we also identified as atomic components.

Furthermore, the role of these atomic components has been recognized to support information hiding and thus, maintainability (Ghezzi et al., 1991) and reuse (Sommerville, 1992). Guttag (1977) argues that ADTs are a good way to design systems because they provide unambiguous specifications that can be used to partition implementation tasks, they can be formally verified, and they postpone implementation decisions, leading to more efficient implementations chosen after more is known about the behavior of the system. This rationale could also be extended to perfective maintenance, once ADTs are recovered and have been cleanly and clearly encapsulated.

1.4. *Purity of atomic components*

The definitions of ADTs, ASEs, and HCs provided above describe the ideal situation in which programmers would always be aware of and respect the encapsulation of these atomic components. In practice, in languages like C, atomic components are seldom captured explicitly and software development does not always exploit them. As a result, their encapsulation is often violated by direct accesses which bypass the accessor functions of the atomic components.

In general, we use the adjective *pure* in front of an atomic component to denote that all accesses to its internal parts proceed through its interface, and the adjective *rough* in front of atomic components which suffer from encapsulation violation. We use the convention that when *no adjective* is in front of an atomic component, it is a rough atomic component. This convention was selected because rough components are much more frequent than pure components among the atomic components identified by a group of software engineers who analyzed these systems manually for our evaluation. Actually, their task involved deciding which functions that access internal elements of a potential atomic component were part of the abstraction and which were not.

The presence of these encapsulation violations should be taken into consideration by reverse engineering techniques that attempt to identify atomic components in an automatic or semi-automatic fashion.

1.5. *Article's focus*

This article presents a reverse engineering approach, called *similarity clustering*, which groups together functions, types, and variables into ADT, ASE, and HC candidates according to a similarity metric. The set of features considered by this metric includes the context of these elements, the relationships to their environment, and informal information. In order to evaluate the proposed similarity clustering approach, it was applied to three C systems and the results were compared with the atomic components identified by software engineers. Five other published atomic component identification techniques were applied to these systems. On this benchmark, our similarity clustering approach identifies, in most cases, a higher number of ASEs and ADTs than the other techniques.

1.6. *Article 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 gives an overview of existing techniques to detect atomic components. Section 4 presents our similarity clustering approach. Section 5 describes the benchmark we used to evaluate the detection quality of automatic techniques. The benchmark is a set of components manually compiled by a group of software engineers. We not only used them in the evaluation experiment described in Section 8, but also to calibrate the parameters of our approach in Section 6. Section 7 describes the parameter values that were used in the experiment. In Section 9, we illustrate how similarity clustering can be applied in practice. Section 10 concludes and proposes further research.

2. A conceptual model for programming language entities

By definition, ADTs consist of a type and some accessor routines. The type must be mentioned in the signature of its accessor routines either as formal parameter or as result type. Similarly, ASEs consist of global variables and accessor routines that set or use these global variables. Hence, we have the following relationships between routines and types or variables, respectively, obtainable by analysis of the source code of a program:

- *parameter-of-type*: A type appears in the formal parameters list of a routine.
- *return-type*: A type is used as the result type of a routine.
- *var-access*: A routine accesses a global variable: either it assigns a new value to this global variable, it reads a global variable's value, or uses its address.

All techniques for detecting atomic components are based upon these relationships. For our new similarity clustering approach described in Section 4, we also investigated the following relationships:

- *is-part-of-type*: A type is used in another type's declaration, e.g., in the C code fragment 'struct S {T x;}', T is a part type of S.
- *calls*: A routine calls another routine.
- *actual-parameter*: A global variable is used as an actual parameter in a routine call.
- *access-fields*: A routine accesses the fields of a given user-defined type.
- *var-of-type*: A global variable is of a given user-defined type.
- *local-var-of-type*: A local variable is of a given user-defined type.

Figure 1 is the entity relationship model for the basic parts of an atomic component, namely the entity types *variable*, *type*, and *routine*, and their relationships just described. An instance of this entity relationship model for a given program is commonly referred to

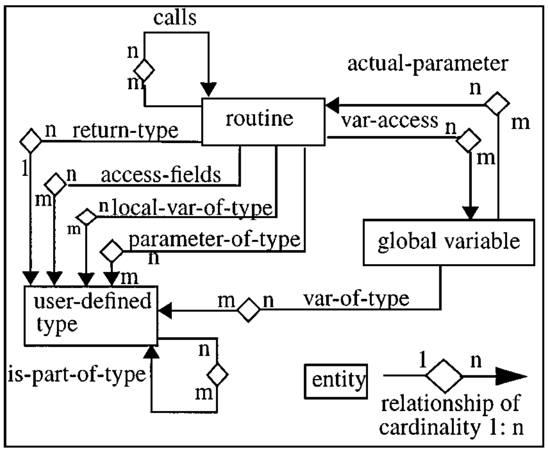

Figure 1. A data model for atomic components identification.

```
/*--- contents of file list.h ---*/
typedef ... T;
struct List     {int length;
                 T contents[100];};
/*--- contents of file list.c ---*/
struct List empty () {
  struct List result;
  result.length = 0;
  return result;
}
void prepend (struct List *l, T t) {
  l->contents [l->length] = t;
  l->length++;
}
```

```
T first (struct List l) {
  return l.contents [l.length-1];
}
struct List rest (struct List l) {
  l.length--;
  return l;
}
```

```
/*--- contents of file stack.c ---*/
static struct List stack;
void init () {
  stack = empty ();
}
void push (T t) {
  prepend (&stack, t);
}
T pop () {
  T result = first (stack);
  stack = rest (stack);
  return result;
}
```

Figure 2. Sample C program.

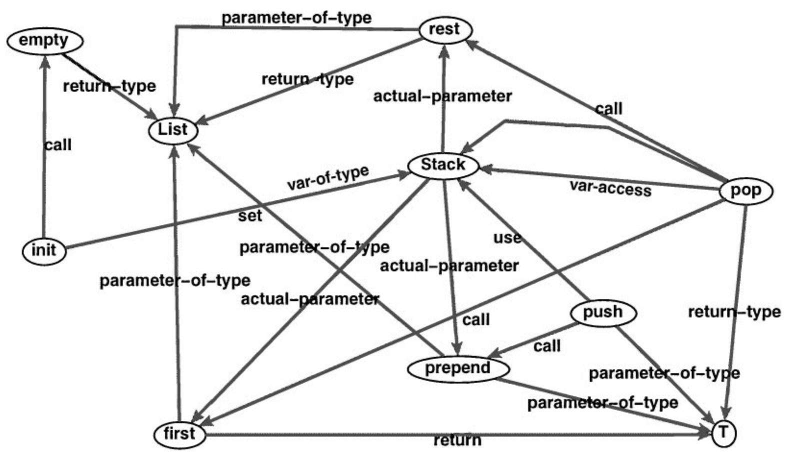

Figure 3. Example resource flow graph.

as *resource flow graph.* It consists of the actual routines, variables, and types and their relationships in a given program. As an example, the resource flow graph for the program in figure 2 is shown in figure 3.

The resource flow graph is used in this article to describe existing approaches to detect abstract data types and abstract state encapsulations. Moreover, it is also used to define the similarity metric at the core of our new similarity clustering method to detect atomic components.

3. Heuristics for detecting atomic components

At a higher level of abstraction, an abstract data type consists of a domain of values for the type and some allowed operations on that type (Sommerville, 1992). In an implementation of an abstract data type, the domain of values is implemented by a data structure which is read and set by routines—its operations. The user of an abstract data type can declare objects of that type and pass them as actual parameters to the operations. Consequently, it is a necessary prerequisite for operations of an abstract data type to mention the data type in their signature, i.e., their parameter list or their return type in the case of functions. That is, all routines with a data type T in their signature are candidates for an operation of the abstract data type T. However, this prerequisite is necessary but not sufficient. Some routines simply pass a value of T to other routines and are not true operations of T. Many routines have more than one parameter type, making it necessary to decide which one they belong to. For all sorts of routines that convert one type to another type this can be very hard to judge. Sometimes—especially in programming languages that do not provide record types such as Fortran77—one even has to look at several basetypes of the underlying programming language in a parameter list to form one abstract data type. For example, one can have a stack implementation that passes the stack contents realized by an array and the stack pointer given as an integer as two distinct parameters.

Similarly, an abstract state encapsulation represents an abstraction of a state and the operations that manipulate the state. The state is implemented by a set of static global variables. These variables are set and used by the operations of the abstract state encapsulation. Most of the time, programmers do not make the effort to group the global variables of an ASE together as components of a record structure to make the connection of the variables obvious. In many old programming languages, they would not even have a chance to do so because these do not support user-defined data types. Even in programs written in modern programming languages, one often finds accesses to these global variables by routines that do not belong to the ASE because of efficiency considerations. All that makes it difficult to find the variables that together make up the abstract state and the routines that really represent the ASE's operations.

Considering these facts, it is obvious that the naive approach of grouping types together with all routines whose signature refers to them, and of aggregating variables with all routines that set or use them, leads to erroneously large candidate components in many cases. This strategy is discussed in the next subsection for global variable references. In the rest of this section, we present heuristics proposed to avoid erroneously large ADT and ASE candidates by going beyond the simple reference criterion. Some of them can detect ADTs as well as ASEs, some of them are specialized on one type of atomic component. Those of them that can detect both types of atomic components merge the results of ASE and ADT detection into a hybrid candidate, if there is a routine that belongs to both an ADT and an ASE.

3.1. Global variable references

Yeh et al. (1995) identify ASEs by grouping global variables and all the routines that access them, regardless of where they are declared. We will refer to this strategy as *Global Variable Reference*. In the case of a global error variable that is used in many parts of the system, this approach will collapse a large part of the system into one atomic component. Yeh et al. propose to exclude frequently used variables from the analysis to avoid this unwanted effect.

Applied to the example of figure 2, Global Variable Reference heuristic would find the ASE {stack, init, push, pop}.

3.2. Same module heuristic

One simple heuristic that follows programming conventions and is easy to apply is to group together only those routines, data types, and global variables that are declared in the same module. In the case of Ada, a package body and its specification would form a module. In C, modules do not exist, but programmers simulate the lacking concept by a header file f.h for the specification and a C file f.c for the body.

The Same Module heuristic (Girard and Koschke, 1997) assumes that programmers are disciplined and follow this convention. If a programmer puts each routine in a separate file, the Same Module heuristic cannot yield any result. Moreover, for modules with several distinct abstract data types with conversion routines between each other, this heuristic groups all those routines and data types together in one large component.

In a comparison (Girard et al., 1997) with most other techniques presented in this section, the Same Module heuristic turned out to yield best results with respect to three reference systems. This heuristic can be applied to detect ADTs as well as ASEs.

For the example of figure 2, the Same Module heuristic would propose one ADT {T, List, empty, prepend, first, last} and one ASE {stack, init, push, pop}. Had those declarations been in one common file, “Same Module heuristic” would have produced one big hybrid component consisting of the elements of the ADT and ASE mentioned.

3.3. Part type heuristic

Often, we find abstract data types that represent some sort of container of other abstract data types. For example, queues are containers of processes, or an account contains data about its owner and the deposited money. For such abstract data types, there is usually an operation that takes an element and puts it into the container. For a process queue, for example, there will be an *insert* routine with two arguments: the process to be inserted and the queue itself. Even though both types are mentioned in its signature, we would not consider *insert* to be an operation for processes but for queues. The Part Type heuristic reflects this perception. It is based on the part type relationship which is defined as:

- a type PT that is used in the declaration of another type T is called a part type of T, denoted $PT < T$
- the part type relationship is transitive; i.e., if $PT < T1$ and $T1 < T$ then $PT < T$ holds

The Part Type heuristic groups a routine with those types in its signature that are not a part type of another type in the same signature. This can be illustrated with the example of figure 2 in which we find the following declarations:

```
typedef . . . T;
struct List {int length; T contents[100];};
void prepend (struct List *l, T t) {
```

Here, T is a part type of *List*. That is why *prepend* would be an operation of *List* according to the Part Type heuristic and not of T though both are mentioned in *prepend*’s signature.

As opposed to the Same Module heuristic, the Part Type heuristic does not rely on the programmer's distribution of routines into modules. However, it assumes that the parameter of a part type is actually used to be put into its container or to be retrieved from it. Since it does not analyze the actual usage any further, it is going to fail if this assumption is false. For example, if the part type relationship exists only by accident, or if the routine just passes the two arguments to an actual operation of the abstract data type, it will consider the routine to be an operation of the abstract data type.

Liu and Wilde proposed this heuristic (Liu and Wilde, 1990). The Part Type heuristic can only be applied for detecting abstract data types.

Because T is a part type of List in the example in figure 2, the Part Type heuristic would detect one ADT consisting of {List, empty, prepend, first, last}.

3.4. Internal access heuristic

The purpose of an abstract data type is to hide implementation details of the internal data structure by providing access to it exclusively through a well-defined set of operations. Liskov and Zilles' idealized definition entails that all routines that access internal components of the abstract data type are considered to be the data type's operations, which is exactly the attitude of the Internal Access heuristic. Internal access for a type T means:

- if T is an array, then any index subscript is an internal access;
- if T is a record, then any field selection is an internal access;
- if T is a pointer, then any dereference is an internal access.
- if T is a base type, any application of a standard operator is considered an internal access since the actual data structure is used non-abstractly

The Internal Access heuristic associates types with those routines that have an internal access to them. Yeh et al. presented this heuristic (Yeh et al., 1995). Originally, they only proposed to consider internal access to record types, but the same can be applied to arrays and pointers as well.

As opposed to the Part Type heuristic, the Internal Access heuristic really checks how the parameter type is used. However, in real programs one often finds Liskov and Zilles' rule violated. This often happens for reasons of efficiency or convenience in the case of data types of which the programmer is convinced that their representation will never change. For example, an abstract data type for complex numbers will always have an imaginary and a real part. That programmers can be mistaken and what consequences this can have, is demonstrated these days by date representations in legacy systems.

Though originally only suggested by its authors for detecting abstract data types, this heuristic can also be used to find abstract state encapsulations in cases in which state is implemented by a record, array, or pointer. We will refer to the strategy to extend the Internal Access heuristic to detect ASEs as *Variable Internal Access* as opposed to Global Variable Reference discussed in Section 3.1.

In the example of figure 2, *empty*, *prepend*, *first*, and *last* would be added by the Internal Access heuristic to the type List because of their internal access to it. Variable Internal Access would not detect the ASE {stack, init, push, pop}, because the routines do not access the internal record components of stack.

3.5. Interconnectivity metric

The approach proposed by Canfora et al. (1993, 1996) uses a usage-pattern-based heuristic to generate ASE candidates. These candidates are then ranked according to an index which measures the variation in “internal connectivity” of the system due to the introduction of these ASEs. The method selects candidates with a value of internal connectivity above a threshold obtained by statistical sampling.

The heuristic and the evaluation metric are defined on a bipartite subgraph of the resource flow graph that describes the usage of global variables by functions. They can be explained more easily in terms of the following definitions, given a function f and a global variable v:

- the *context of f* is the set of all variables it sets or uses
- *functions related to f* are all functions which set or use variables in the context of f
- *closely-related functions of f* are all functions which set or use *only variables* in the context of f
- *functions referencing* v are all functions which set or use v

Canfora’s heuristic generates a candidate for each function f. A candidate consists of:

— the context of f
— closely-related functions of f

The candidates are ranked according to ΔIC which is defined as:

$$\Delta\text{IC}(F) = \frac{|\text{closely related functions of } F|}{|\text{functions related to } F|} - \sum_{v \text{ in context of } F} \frac{|\{f' \mid \text{context of } f' = \{v\}\}|}{|\text{functions referencing } v|}$$

This formula can be read as the internal connectivity of the candidate built around F minus the internal connectivity of all trivial components. Each of these trivial components contains a single variable.

The original approach presented in Canfora et al. (1996) uses the following algorithm:

```
repeat
  build variable-reference graph
  create cluster candidates using a heuristic
  for each candidate
    compute improvement in cohesion metric ($\Delta$IC) introducing
    these candidates
    if improvement > = threshold then
      select candidate
    else
      slice remaining functions using different clusters’ variables
until  graph contains only isolated subgraphs consisting of one
       variable grouping with one or more functions
```

The current article leaves out the slicing step (removing the slicing step reduces this approach to considering only one iteration), because this would modify the functions of the resulting system, hence making the comparison with manual results and results from other techniques more ambiguous.

For the experiment described in Section 8, the reported thresholds were established by running the prototype with all threshold values between 0.1 and 0.9 by increments of 0.1; identifying the threshold that produced the best results; and performing more runs in intervals of 0.1 centered on this threshold using increments of 0.01. The threshold corresponding to the best results is reported.

3.6. AC-arch

Schwanke proposed a clustering of routines into modules based on a similarity metric (Schwanke, 1991). The tool which implemented his approach, *Arch*, aimed at detecting subsystems and therefore he only considered routines. This approach was extended to detect atomic components in Girard et al. (1997). It considers routines, types, global variables, and relationships among them (see Section 2). The clustering algorithm used in this approach works as follows:

```
place each routine in a group by itself
repeat
    identify the most similar groups
    combine them
until the existing groups are satisfactory
```

In each iteration, the most similar groups are combined. This group similarity is based on a similarity between simple entities, so it is explained first. These simple entities are routines, global variables, and types. Given two simple entities A and B, the similarity metric used during clustering is defined as follows:

$$Sim(A, B) = \frac{Common(A, B) + k \times Linked(A, B)}{n + Common(A, B) + d \times Distinct(A, B)}$$

wherein *Common(A, B)* reflects the number of common attributes of A and B and *Distinct (A, B)* reflects the number of distinct attributes. Attributes of A are all programming language entities (presented in figure 1) that have an edge to or from A in terms of the resource flow graph.

Linked(A,B) is 1 if A shares an edge with B, otherwise it is 0. The two parameters $k \geq 0$ and $d \geq 0$ are weights given to *Linked* and *Distinct* in *Sim*. They have to be ascertained by experiments on a sample of the subject system. The parameter $n \geq 0$ is used for normalization purposes, in this article it is set to 0.

The similarity between components to merge is computed in each iteration of the clustering algorithm as $Sim(A \cup B, X) = \frac{|A| - Sim(A,X) + |B| - Sim(B,X)}{|A|+|B|}$.

This technique yields ADTs and ASEs. In fact, it can also retrieve groups of related functions such as mathematical functions of a math library. This way, our extension of

Schwanke's similarity is more general than all other techniques presented so far. However, in the context of this article we are only interested in abstract data types and state encapsulations. We name this adaptation of the Arch approach to the recovery of atomic components *AC-Arch*.

3.7. AC-iArch

In Schwanke and Hanson (1994), Schwanke and Hanson proposed a refined version of their Arch approach/tool which they call *iArch*. In iArch, they used an ordinal measure of similarity instead of a interval one and exploit a nearest neighbor approach to classify components. They address the problem of learning the appropriate weights using a neural network. They use jackknife testing to evaluate the predictive power of learned parameters. Adapting this approach to identify atomic components is part of our future work.

3.8. ASE identification by conception formation methods

This issue of the journal also presents work by Sahraoui et al. who use concept formation methods to detect ASEs (they call them objects). It consists of three steps. First, the Galois lattice is built for the variable reference graph, a subgraph of the resource flow graph that consists solely of variables and routines. Second, in the Galois lattice, state variables of ASEs are identified, using a heuristic that is based on how often variables are used simultaneously thereby preferring smaller sets of variables. Third, the routines that belong to those ASEs are identified. If a routine only uses variables in one ASE, i.e., a set of related state variables identified in the previous step, it belongs to this ASE. If it uses state variables of different ASEs, but sets only variables in exactly one ASE, it is grouped with the latter. And finally, if the routines sets and uses variables of different ASEs, they propose to slice the routine.

4. Similarity clustering approach

This section presents the new similarity clustering approach. It first gives an overview of the approach. It then discusses the various aspects of similarity that are combined in the approach.

This approach is inspired by Schwanke's work (Schwanke, 1991) aimed at detecting subsystems using a similarity metric between routines (see Section 3.6). The similarity clustering approach applies the idea of this work to atomic components identification by generalizing the similarity metric, adding informal information, edge-dependent weights, and adapting many of its parameters. The two approaches will be contrasted in more detail in Section 4.5.

4.1. Overview of the approach

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.

To form atomic component candidates, 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 that are shared. The algorithm terminates when "existing groups are satisfactory". In the experiment reported in this article, groups are considered satisfactory when the most similar groups have a similarity that 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 two groups of entities S_1 and S_2 is defined as the average of the similarities of all pairs of entities in the two sets:

$$GSim(S_1, S_2) = \frac{\sum_{(s_i \in S_1, s_j \in S_2)} Sim(s_i, s_j)}{size(S_1) \times size(S_2)} \tag{1}$$

We also tried Schwanke's suggestion to use the maximal individual similarity of elements in the two groups. But we experienced that this has the effect of creating very large groups which was not very useful.

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 article:

- Direct relations
- Indirect relations
- Informal information

Direct relations are relations between the two entities compared (see figure 1). Indirect relations are relations with common third entities. Informal information is the information in the program source code that 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 weights x_i are used to adjust the influence of the diverse specific similarities):

$$Sim(A, B) = \frac{x_1 \cdot Sim_{indirect}(A, B) + x_2 \cdot Sim_{direct}(A, B) + x_3 \cdot Sim_{informal}(A, B)}{x_1 + x_2 + x_3} \tag{2}$$

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

4.2. Direct relations

Direct relations represent immediate connections between two entities. Their contribution to the similarity is computed as $Sim_{direct}(A, B)$, which yields a value between 0 and 1. This is defined as the weighted sum of edges between A and B divided by the weighted sum of all possible edge types between the type of node A and the type of node B in the resource flow graph (see figure 1):

$$Sim_{direct}(A, B) = \frac{W(link(A, B))}{W(\text{all-links}(nodetype(A), nodetype(B)))} \tag{3}$$

where $link(A, B)$ denotes the actual links between A and B, and *all-links()* denotes all edge-types that are possible in a resource flow graph between two given node types.

$$W(X) = \sum_{x \in X} weight(x) \tag{4}$$

where $weight(x) \geq 0$ is a weighting factor that allows assigning certain features more influence on the global value of the metric.

Weights. The weight factors are introduced to give more influence to certain features. There are several alternatives for their definition.

Shannon's information content: Schwanke proposed to use Shannon's information content (Shannon, 1972) from information theory as weighting factor *weight(x)*:

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

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 kinds 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. How one can find edge weights by a statistical analysis is shown in Section 6.

Combined strategy: Shannon's information contents and edge type weights can be combined by multiplying the two values. This strategy unifies both advantages: it allows tuning clustering for specific patterns and makes frequently occurring entities less important. The results reported in this article are based on this combined strategy.

4.3. 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:

$$Sim_{indirect}(A, B) = \frac{Common(A, B)}{Common(A, B) + d \cdot Distinct(A, B)} \tag{5}$$

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 that regulates the importance given to distinct features. In terms of the resource flow graph, they are defined as follows:

$$\begin{aligned} Common(A, B) &= W(edges\text{-}with(A) \cap edges\text{-}with(B)) \\ Distinct(A, B) &= W(edges\text{-}with(A) \oplus edges\text{-}with(B)) \end{aligned}$$

The operator $\oplus$ denotes the symmetric difference for sets. The term *edges-with(B)* refers to the set of pairs (E, N) where N is a node connected to B via an edge of type E. These pairs are weighted by the combined strategy of Shannon's information content and fixed edge weights.

4.4. 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 to 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 (Biggerstaff, 1989)), which focus on the information derived 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. The following formula is used to compute the informal similarity between entities:

$$Sim_{informal}(A, B) = \frac{\sum_{t \in T} x_t \cdot Sim_t(A, B)}{\sum_{t \in T} x_t} \qquad T = \left\{ \begin{array}{c} words \\ suffix \\ filename \end{array} \right\} \tag{6}$$

4.4.1. 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 (Biggerstaff, 1989) 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., f1 instead of top_stack). The naming of identifiers also conveys 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:

$$Sim_{words}(X, Y) = \frac{|\mathrm{words}(X) \cap \mathrm{words}(Y)|}{|\mathrm{words}(X) \cup \mathrm{words}(Y)|} \tag{7}$$

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

$$Sim_{suffix}(X, Y) = \begin{cases} 1 & X = Y \\ \dfrac{\mathrm{prefix}(X, Y) + \mathrm{postfix}(X, Y)}{1 + \mathrm{prefix}(X, Y) + \mathrm{postfix}(X, Y)} & X \neq Y \end{cases} \tag{8}$$

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.

4.4.2. 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*):

$$Sim_{filename}(X, Y) = Sim_{suffix}(\,fname(X),\ fname(Y)) \tag{9}$$

4.5. Differences from previous approaches

In Schwanke (1991), Schwanke defines a similarity metric used to group functions into modules (see Section 3.6). Our approach, like Schwanke's approach, differs from the other approaches presented so far, in that it generates candidates using a metric that exploits a more extensive set of relationships among program entities. The previous approaches compute connected components or other fixed patterns in a graph. In contrast, whether these patterns are present or not, both metric approaches deal with continuous values which reflect the degrees of similarity.

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

Table 1. Differences to Schwanke's approach.

	Schwanke's approach	Article's approach
Domain	Routines	Routines User-defined types Global variables
Weights	Shannon info.	Shannon info. Edge weights reflecting the type of relation between entities
Features	Usage of non-local names	Type relations Usage relations Call relations
Informal information	Not used	Tokens in identifiers Pre- and postfix in identifiers Organization of files
Direct links	Values = 0 or 1	Continuous values between 0 and 1
Similarity between groups	Maximum similarity between elements	Average similarity between elements

4.6. Related research on similarity metrics

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 (1994) combines similarity metrics with domain knowledge to provide explanations of case reasoning. However, some references like Richter (1992) present a good overview of the basic ideas and introduce learning approaches that can be applied to discover the appropriate similarity. A systematic search for the best possible similarity metric is beyond the objective of this paper. However, it is future work to identify what could be applied from related work on similarity metrics.

5. Reference components

In order to establish a comparison point for the detection quality of the automatic recovery techniques, a group of software engineers manually compiled a list of *reference atomic components* for three systems. For the evaluation, we compared the components proposed by automatic techniques, called *candidate atomic components*, to the reference components. This section summarizes how the reference components were obtained and validated. The experiment that uses these reference components to evaluate our similarity in concert with other published approaches is discussed in Section 8. In Section 6 we describe how the reference components were exploited to find the edge weights used for the experiment.

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-based simulator for rigid body systems

Table 2. Suite of analyzed C systems.

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

Table 3. Number of atomic components in analyzed systems.

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

(Keller et al., 1995), bash is a Unix shell, and CVS is a tool for controlling concurrent software development.

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 its atomic components.

Details concerning the guidelines given to the software engineers, their experience, and how the task was divided among them are described in Girard et al. (1997). 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 system studied.

5.3. *Reference components quality*

The fact that our reference atomic components used as comparison point were produced by people raises two questions: Whether other software engineers would identify the same atomic components and how much their opinion would differ. This section describes how these questions were investigated.

5.3.1. Acceptable oracle. The first step we took was to define when a set of reference atomic components can act as an acceptable oracle against which the results of techniques can be compared. The following assumptions are central to the notion of oracle we used:

1. One assumption is that the oracle does not have to be complete to be acceptable. That is to say, that it does not contain all possible atomic components that some particularly creative software engineer could find, but a reasonable subset of those that would be considered correct by most software engineers.

2. On the other hand, the oracle should not contain any false atomic components, that is, atomic components which would be considered incorrect by most software engineers.

5.3.2. Multiple software engineers. In order to investigate whether multiple software engineers would identify the same atomic components, we performed an experiment on a subset of CVS containing 2.8 Kloc. This subset is composed of the following key files: history.c, lock.c, cvs.h. These source files were distributed along with a cross-reference table indicating the relation among types, global variables, and functions. Four software engineers were given the task of identifying the atomic components present. We collected a description of the procedure they followed along with their results, then looked for cases where they seemed to have broken their own rules and asked them to refine either their procedure or their results. We also revisited with them those atomic components where a comment indicated that they were unsure or something was unclear and corrected their results according to their conclusions.

The four software engineers agreed on the basic principles that characterize an atomic component. There were some divergences on the details, for example one of them added to ADT functions that did not have the type T of the ADT in their signature, but applied a cast of type T to one of their parameters. These divergences occurred rarely.

5.3.3. Level of agreement. Given four versions of the atomic components that were found by different software engineers, the next question is how to compare these versions in such a way as to quantify the level of their agreement. Before constructing a formula to evaluate this agreement, we need to define the conditions under which the results of two software engineers agree:

1. The fact that in certain contexts, overlaps of atomic components are meaningful and the fact that no directive excluding overlaps was given to the software engineers lead us to accept overlaps in the reference components.
2. When two software engineers recognize an atomic component, they should agree, to a high degree, on the types, variables, and functions it contains. Therefore, when two atomic components share *20% or less of the union*[1] of their elements, they should be *considered unrelated* and should not be taken into account when computing the agreement level between software engineers.
3. It is possible to describe certain complex atomic components at different levels of granularity. To accommodate this possibility, when comparing an atomic component that has been decomposed into many atomic components by another analyst, the large atomic component should be compared to the combination of all the corresponding decomposed atomic components.

The proposed measure of agreement between sets of atomic components reflects these rules and the assumptions for an acceptable oracle. It is constructed using an intuitive notion of agreement between two sets, which is captured by the following formula:

$$\alpha(a, b) = \frac{|a \cap b|}{|a \cup b|}$$

At the basic level, the agreement measure compares two atomic components, a and b. The *simple-agreement* between them is defined to be:

- 0 if a and b are unrelated (i.e., $\alpha(a, b) \leq 20\%$), because of rule 1
- $\alpha(a, b)$ if a and b are related $\alpha(a, b) > 20\%$

To reflect the different level of granularity used by different software engineers (rule 2), it is necessary, in some cases, to compare a single large atomic component (AC) to a group of ACs that corresponds to a decomposition of the larger one. In order to perform such comparison with a larger AC a, the agreement measure has to select the appropriate group of smaller ACs s_i from all the components which overlap a. The smaller ACs s_i should share a large proportion of their elements with a (using $\rho(a, s_i) = |a \cap s_i|/|s_i|$ as measure). For this evaluation, this proportion selected is 70%, because for many ADTs that contain 3-5 functions, a single mismatched function would imply 70% sharing. The group of all the relevant small components s_i are combined $\Gamma(a) = \cup\, s_i\ \forall s_i\, \rho(a, s_i) \geq 70\%$ and compared to a. The *decomposition-agreement* of a is defined to be:

- 0 if a and $\Gamma(a)$ are unrelated (i.e., $\alpha(a, \Gamma(a)) \leq 20\%$).
- $\alpha(a, \Gamma(a))$ if they are not unrelated (i.e., $\alpha(a, \Gamma(a)) > 20\%$).

The agreement between the sets of ACs (A, B) identified by two software engineers combines the notions introduced above. It combines the *simple-agreement* of simple ACs of A and B, taken pairwise and the *decomposition-agreement* of each single AC which is decomposed into multiple smaller ACs in the other reference set. In both cases, the agreement is divided by the number of cases where the agreements are not unrelated. This agreement formula, named *reference-set-agreement* (A, B), is computed as the sum of the following terms:

- $\sum \alpha(a, b)/|\alpha(a, b) > 20\%| \quad \forall a \in A, |\Gamma(a)| \leq 1, \forall b \in b, |\Gamma(b)| \leq 1, \alpha(a, b) > 20\%$
- $\sum \alpha(a, \Gamma(a))/|\alpha(a, \Gamma(a)) > 20\%| \quad \forall a \in A, |\Gamma(a)| \geq 2, \alpha(a, \Gamma(a)) > 20\%$
- $\sum \alpha(b, \Gamma(b))/|\alpha(b, \Gamma(b)) > 20\%| \quad \forall b \in B, |\Gamma(b)| \geq 2, \alpha(b, \Gamma(b)) > 20\%$

The reference-set-agreements obtained for the four versions of the atomic components identified by different software engineers is presented in Table 4.

For these software engineers, the average reference-set-agreement is 0.75. As a comparison point four versions of atomic components—which would contain 3 elements each, of

Table 4. Agreement among software engineers.

	S.E. 1	S.E. 2	S.E. 3	S.E. 4
S.E. 1	▬	0.739	0.493	1.000
S.E. 2	0.739	▬	0.700	0.790
S.E. 3	0.493	0.700	▬	0.800
S.E. 4	1.000	0.790	0.800	▬

which 2 would exactly agree and 1 would share half of their element with the others—would have an average reference-set-agrement of 0.76. The measure seems to be almost unaffected by the number of reference components, as the same pattern repeated over eight atomic components produce an average reference-set-agreement of 0.77. Given these agreements, we conclude that the reference components are a suitable comparison point.

6. Statistical analysis of edge distribution

As described in Section 4.2, there are two ways of obtaining the weights for direct and indirect similarities, i.e., for the weighted sum $W(x)$ used in formulas (2) and (3). Schwanke proposes to use Shannon's information content for clustering routines into subsystems. In his approach, the importance of a feature is only based on how often it is used. This can be sufficient for clustering routines. However, the atomic components follow more specific patterns. For an ASE, for example, it is intuitively clear that uses and sets are more significant than routine calls. This suggests a model in which it is possible to give individual weights to edge types. In order to answer the questions of whether our intuition is correct at all and how one can find appropriate edge weights, we investigated the distribution of edge types in the reference components.

Because only a portion of all entities in the subject systems really belongs to an atomic component (according to our analysts), considering a global analysis of edge distribution does not make sense. Instead, for each edge type we compute the ratio of edges completely within an atomic component and those that cross the border of an atomic component. Let AC denote the set of all reference atomic components. Then, more precisely, we calculate for each edge type E:

$$EdgeRatio_E = \frac{1}{|AC|} \times \sum_{A \in AC} \frac{inside_E(A)}{inside_E(A) + across_E(A)} \tag{10}$$

where $inside_E(A)$ is the number of edges of type E for which both ends are in A, and $across_E(A)$ are those edges of type E that have exactly one end in A. This can best be illustrated in figure 4.

Solid and dashed edges represent different edge types, namely calls and uses, respectively. The atomic component itself consists of {v1, v2, f3, f5, f6}. Then $inside_E(call)$ is 1 because there is only one call from f6 to f3 completely inside the atomic component. $Across_E(call)$ is 2 because f2 calls f5 and f6 calls f4.

In the following section, we will refer to the set of edges of an atomic component that are *inside* or *across* as its *context*. Tables 5 and 6 report, for Aero, Bash, and CVS, the edge

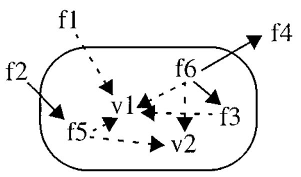

Figure 4. Example context of an atomic component.

Table 5. Edge ratios (white) and frequency (shaded) for ADT.

Edge type	Aero		Bash		CVS	
Call	0.07	693	0.05	680	0.12	504
Variable-set	0.0	35	0.00	42	0.00	0
Variable-use	0.0	95	0.00	62	0.00	12
Variable-address	0.0	3	0.00	5	0.00	0
Field-set	0.51	73	0.12	39	0.08	13
Field-use	0.85	861	0.68	286	0.48	205
Field-address	0.43	48	0.00	0	0.08	5
Parameter-of-type	0.42	265	0.57	155	0.59	89
Return-type	0.04	31	0.65	71	0.21	19
Local-var-of-type	0.13	174	0.29	145	0.03	119
Actual-parameter	0.0	18	0.00	17	0.00	78
Is-part-of-type	0.27	77	0.31	71	0.16	76

Table 6. Edge ratios (white) and frequency (shaded) for ASE.

Edge type	Aero		Bash		CVS	
Call	0.03	602	0.04	662	0.04	1724
Variable-set	0.58	130	0.38	89	0.85	308
Variable-use	0.51	221	0.40	138	0.83	546
Variable-address	0.28	9	0.00	3	0.46	59
Field-set	0.11	37	0.00	10	0.03	38
Field-use	0.20	267	0.01	60	0.02	299
Field-address	0.15	27	0.00	0	0.00	1
Parameter-of-type	0.00	166	0.00	18	0.00	58
Return-type	0.00	5	0.00	4	0.00	6
Local-var-of-type	0.00	68	0.00	53	0.00	170
Actual-parameter	0.00	168	0.00	47	0.03	742
Is-part-of-type	0.00	0	0.00	0	0.00	0

ratios for ADTs and ASEs, and the absolute frequency of the edges in the ADT's and ASE's context. Note that all edge ratio values are rounded up; a ratio of 0.00 actually means that the ratio is below 0.005.

An edge ratio around 0.5 indicates that the corresponding relationship has little influence; an edge ratio far below 0.5 states that the relationship should be disregarded, whereas an edge ratio far above 0.5 means that it is important to the respective kind of atomic component.

As expected, set and use are most important for ASEs and return-type and parameter-of-type are most important for ADTs. Set is more important than use. A similar conclusion for return-type and parameter-type cannot be drawn. As we have seen, the influence of return-type can even be rather weak, depending on the system. A promising future extension would also be to take the quality of the parameter-type and return-type relationships into account, i.e., to distinguish between types that are only passed to other routines and types whose internals are accessed (incorporating the Internal Access heuristic).

All other edge types have very limited relevance. Another observation is that most edge type ratios are not stable across systems, which causes the necessity of finding individual edge weights for different systems. In this article, we considered very different kinds of systems from different authors. For a family of systems from a common application domain and for programmer teams with established programming conventions, we expect less divergence of edge weights.

Conclusions for the edge weights. The edge weights contribute to the direct and indirect similarity between two entities. A high value has the effect of attracting two entities during clustering. Therefore, the likelihood of them being in the same atomic component increases. It makes good sense to use the edge type ratios as edge weights, because they tell something about the "natural" consistence of an atomic component. Using a high edge ratio for set edge weights draws many set edges into the atomic component, while a low edge ratio for actual-parameter has only little attraction.

Because we were only interested in the characteristics of ADTs and ASEs, we considered only the contexts of these atomic components. However, focusing on atomic component contexts can lead to wrong conclusions if the global edge frequency differs from the local one. With global edge frequency of an edge type E we mean E's share of the overall set of edges. The local edge frequency of E is the frequency of E in the context of atomic components. We compared global with local frequencies for the subject systems and experienced that they indeed differ for some edge types. If the global frequency equals the local frequency, the edge ratio can be used as is. If the global frequency is less than the local one, then there are more edges of E in the context of atomic components than in the rest of the system. If this is the case, we have all the more reasons to assign high edge weights for edge types with high ratios to increase their attraction, hence reinforcing creation of atomic components in this situation. In the case of a global frequency greater than the local one, the situation reverses. There are less edges of this type in the context of atomic components and therefore the edge weights should be decreased to avoid false positives to be created elsewhere. The following formula for edge weights based on edge ratios reflects this:

$$weight_E = \frac{\text{local-frequency}_E}{\text{global-frequency}_E} \times EdgeRatio_E \tag{11}$$

This value can then be combined with *Shannon's information content* as discussed in Section 4.2. Note that *Shannon's information content* takes the frequency of nodes into account and does not interfere with the edge type frequency.

Of course, this is just a reasonable heuristic, not a real mathematical prediction. The many other parameters that influence similarity between entities, the way we define similarity between groups of entities in terms of similarity of single entities, and, in particular, the iterative nature of the clustering algorithm makes it very difficult to model the approach mathematically. We are currently investigating how we can apply statistical methods to establish the other parameters of the approach.

Table 7. Investigated ranges of parameters.

	x_{indirect}	x_{direct}	d
Aero	2–4	0.5–3	0–3
Bash	2–7	0.5–4	0–5
CVS	2–11	0.5–7	0–5

Table 8. Parameters used to produce reported results.

		x_{indirect}	x_{direct}	x_{informal}	x_{suffix}	x_{filename}	x_{word}	d
Aero	ADT	2.0	1.0	2.2	0.2	1.0	1.0	3.0
	ASE	3.0	0.5	2.2	0.2	1.0	1.0	3.0
Bash	ADT	6.0	2.5	2.2	0.2	1.0	1.0	1.0
	ASE	1.0	2.0	2.2	0.2	1.0	1.0	1.0
CVS	ADT	5.0	5.0	2.2	0.2	1.0	1.0	0.0
	ASE	2.0	5.0	2.2	0.2	1.0	1.0	2.5

7. Similarity clustering weights and parameter settings

We selected the following parameters and weights for our similarity clustering approach (see Section 4.1). They were used to produce the results in the experiment presented in next section. We used 0.2 as clustering threshold for all systems. We chose edge weights according to (11) and multiply them by Shannon's information contents to obtain the weighted sum factors $W(x)$. For the other parameters we tried combinations of the following ranges and calibrated them against the reference components (Table 7).

In earlier experiment, we observed that x_{informal}, x_{suffix}, x_{filename}, and x_{words} lead to reasonable results in all systems, so we kept them fix. However, we calibrated x_{indirect}, x_{direct}, and d for each system. For the experiment, we used the parameter values given in Table 8.

Note that the numbers given here differ from those we reported in Girard et al. (1997) in that we added the edge access-field between routines and type nodes and that we chose different edge weights (gathered by the analysis described in Section 6).

8. Experiment

This section compares the atomic components recovered by similarity clustering with those recovered by other techniques against those identified by software engineers. It explains the comparison method, discusses its outcome, and reports on the false positives.

8.1. *A comparison of candidate and reference components*

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 a match of another component T (denoted by $S \ll T$) if at least 70%[2] of the elements of S are also in T (i.e., $|S \cap T|/|S| \geq 70\%$).

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., $C \ll R$ and $R \ll C$).
  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 and possibly multiple candidates (C_i) or multiple references (R_i) need to be combined:

  —$C_i \ll R$, but not $R \ll C_i$ $(i > 0)$. This case is denoted as $n \sim 1$.
  —$R_i \ll C$, but not $C \ll R_i$ $(i > 0)$. This case is denoted as $1 \sim n$.

  Partial matches of this type require more attention to split, combine, or refine candidate components that were too fine grained or too large.
- All others are *bad* candidate components. They are not close enough to the reference components to guide the software engineer's work. We denote this case an $n \sim m$.

8.2. *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 using the following formula:

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

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

8.3. *Benchmark results*[3,4]

We applied the techniques listed in Table 9 and our new similarity clustering to recover atomic components of the three systems described in Section 5.1.

Note that some of the techniques can only be applied for detecting one type of atomic component, some can detect both ADTs and ASEs (Table 10).

This has to be taken into account when looking at the results. Yeh et al. (1995) used Internal Access to detect ADTs and Global Variable Reference to detect ASEs. In the comparison, we introduced the pure Internal Access approach, which applies Internal Access both to detect ADTs and ASEs, since, in principle, there is no reason why Internal Access should not be applied to detect ASEs.

Table 9. Evaluated combinations of atomic component detection techniques.

Method	ADT detection technique	ASE detection technique
Same module	Same module	Same module
Pure internal access	Internal access	Variable internal access
Part type	Part type	—
Delta-IC	—	Delta-IC
AC-arch	AC-arch	AC-arch
Similarity clustering	Similarity clustering	Similarity clustering

Table 10. Detected ADT and ASE.

		ADT						ASE					
				OK						OK			
		Good		Too large		Too detailed		Good		Too large		Too detailed	
Method	System	#	Acc	#	Acc	#	Acc	#	Acc	#	Acc	#	Acc
Same module	Aero	3	0.75	0	0	1	0.34	6	0.91	3	0.15	2	0.41
	Bash	2	0.78	0	0	0	0	5	0.86	4	0.29	3	0.57
	CVS	5	0.84	1	0.23	5	0.44	30	0.81	3	0.40	9	0.51
Pure internal access	Aero	1	0.91	1	0.31	0	0	2	0.76	2	0.26	2	0.26
	Bash	8	0.79	3	0.33	2	0.57	3	0.83	2	0.39	1	0.29
	CVS	4	0.93	1	0.20	6	0.37	4	0.93	0	0	24	0.47
Part type	Aero	1	0.70	2	0.38	1	0.05	—	—	—	—	—	—
	Bash	8	0.77	2	0.45	1	0.50	—	—	—	—	—	—
	CVS	4	0.94	4	0.31	7	0.32	—	—	—	—	—	—
Delta-IC	Aero	—	—	—	—	—	—	1	0.80	2	0.37	0	0
	Bash	—	—	—	—	—	—	3	0.86	1	0.56	3	0.37
	CVS	—	—	—	—	—	—	10	0.90	1	0.33	14	0.51
AC-arch	Aero	0	0	1	0.57	1	0.50	1	0.68	1	0.57	3	0.36
	Bash	3	0.76	0	0	8	0.41	1	0.8	0	0	6	0.34
	CVS	3	0.85	0	0	7	0.31	6	0.79	0	0	16	0.35
Similarity clustering	Aero	0	0	3	0.48	2	0.43	5	0.91	8	0.42	0	0
	Bash	6	0.66	10	0.39	3	0.55	3	0.81	2	0.42	8	0.34
	CVS	3	0.80	6	0.55	8	0.42	18	0.80	5	0.44	30	0.56

For the comparison to be fair, we must mention that the reference components were used to calibrate similarity clustering, in obtaining the reported results. In practice, one does not have the reference components in advance and has to estimate parameters based on a manually extracted list of reference components of a representative sample of the system. That is, in practice the results can be worse.

Table 11. Number of false positives and missed references.

Technique		Aero False positives	Aero Missed references	Bash False positives	Bash Missed references	CVS False positives	CVS Missed references
Same module	ADT	18 (8)	7	19 (10)	21	7 (4)	7
	ASE	17 (16)	4	56 (43)	9	6 (3)	1
P. internal access	ADT	4 (0)	4	16 (8)	10	12 (5)	6
	ASE	25 (17)	10	63 (32)	15	14 (8)	15
Part type	ADT	16 (7)	1	26 (12)	7	11 (7)	8
	ASE	—	—	—	—	—	—
Delta IC	ADT	—	—	—	—	—	—
	ASE	29 (29)	11	52 (52)	14	4 (4)	17
AC-arch	ADT	22 (18)	9	33 (25)	19	28 (15)	11
	ASE	47 (31)	12	55 (30)	17	33 (26)	20
Similarity clustering	ADT	21 (18)	6	32 (31)	4	37 (30)	3
	ASE	44 (36)	4	57 (54)	7	26 (21)	0

The number of atomic components and their accuracy are not the only aspects to consider, one has to look at the number of false positives (bad category) and the number of reference components which were not recognized by a technique. For the number of false positives, we also report, in parenthesis, the number of false positive, containing more than two elements. These correspond to real false positives as atomic components almost always contain more than two elements and it is easy to filter out candidates which contain only two elements.

As Table 11 reports on these, number similarity clustering produces systematically more false positives, but also usually misses fewer references.

In Table 12, we summarize the quality of the atomic components identified by the various techniques. The table entries correspond to the sum of the accuracy of components (good + OK) recovered by each technique, divided by the number of identified and missed components:

$$\text{Detection\_Quality} = \frac{\sum_{g \in GOOD} accuracy(g) + \sum_{g \in OK} accuracy(g)}{|GOOD| + |OK| + |\text{missed-references}|} \tag{12}$$

It can be observed in Table 12 that similarity clustering has a better detection quality for ADTs and ASEs than the other techniques in most of the cases. In the other cases, its detection quality is second best. However, this comes at the price of additional false positives.

9. How to apply similarity clustering in practice

As with any metric-based clustering approach, the parameters of similarity clustering have to be adjusted before the approach can be applied. In practice, one can select a representative

Table 12. Detection qualities of atomic component detection techniques.

	Aero		Bash		CVS	
Technique	ADT	ASE	ADT	ASE	ADT	ASE
Same module	0.24	0.45	0.07	0.34	0.37	0.77
Part type	0.30	—	0.42	—	0.32	—
Pure internal access	0.20	0.16	0.37	0.17	0.36	0.35
Delta IC	—	0.26	—	0.20	—	0.40
AC-arch	0.10	0.13	0.19	0.12	0.22	0.25
Similarity clustering	0.38	0.46	0.42	0.30	0.59	0.66

sample of the system, then obtain the atomic component in this sample either as identified by an analyst or as reported by one of the automatic techniques listed above. Using these atomic components and the analysis proposed in Section 6, one can derive the edge weights (see formulas (10) and (11)). Using these edge weights and using the fixed informal parameter values proposed in Section 7, one can calibrate the three other parameters of similarity clustering against these atomic components by applying a simple search approach and the detection quality as an evaluation function. With the resulting parameters, similarity clustering is then applied on the whole system. Browsing and evaluating the proposed candidates, one can decide if the results are acceptable or if the parameters need to be refined. To refine the parameters, one can add the satisfactory candidates identified while browsing to the training set and calibrate the parameters once again.

In this calibration, most of the steps are automatic. The user's task is to evaluate reference components and to decide on the parameter ranges to explore while finding a good parameter setting. This constitutes significantly less work than investigating a large software system by hand. Furthermore, when dealing with multiple systems of the same context (same application domain, same organization and using similar programming conventions), it is likely that the setting from one system will require only a small amount of tuning to be reused for another system.

10. Conclusions

In this article, we presented similarity clustering, a new approach to extract two types of *atomic components* (abstract data types and abstract state encapsulations) from source code. At the core of our work is a clustering method based on a similarity metric between elementary program entities that form atomic components (i.e., routines, types, and variables) inspired by the work of Schwanke (1991). 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 effectiveness of this approach in a context where it is important to recover as many correct components as possible. We compared the components identified by similarity clustering and five other published approaches against the components identified by a group of software engineers on three C systems (30-38 Kloc). In most of the cases, our similarity clustering approach has a higher detection quality and identified more ADTs and ASEs than the other techniques. In the other cases, its detection quality was second best.

However, the goal of recognizing atomic components of the architecture of software systems is only partially achieved. Though giving better results than other methods proposed in the literature, our clustering method recovered 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 that are to be reviewed by software engineers. Reviewing candidates is much faster than looking for the atomic components in the source code. Finding atomic components only supported by either text-based or graphical cross reference tools (Rigi) in the code took, on average, 20 h for each of the systems studied, whereas automatic similarity clustering yields candidates in a prototypical implementation on a Sun Sparc Ultra-2 (166 MHz) in about 15 min (which have to be validated afterwards).

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 article could help him make 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 them.

Future work. We plan to further investigate this work along the following three axes: improve our similarity measure, evaluate its predictive power, and develop means for reducing the number of false positives in the candidates produced.

To improve the quality of our similarity measure, we focus on the assignment of appropriate weights for parameters and the various aspects of similarity for software elements using statistical analysis and user validation. One of our goals is to distinguish the relations between aspects that are stable across systems from those which are system-dependent and hence have to be configured for each system. We also intend to explore ordinal measure and nearest-neighbor classification along a line similar to Schwanke (Schwanke and Hanson, 1994) in future similarity-based approaches.

To evaluate similarity clustering's predictive power, we will study the quality of atomic components produced when the parameters and weights are derived from the atomic components identified by an analyst or by other techniques on a subset of the system.

To reduce the number of false positives, we are currently exploring two strategies: combining the various approaches for identifying atomic components and involving an analyst in the recovery process. The analyst is offered an interactive tool implementing these and other techniques. His role is to select the appropriate techniques, decide on the order in which to apply them, select high-level operators to combine their results, and validate the candidates proposed by the techniques. The analyst can even suggest additional atomic components, allowing him to combine his knowledge of the system with the results of the techniques. This work is being explored as part of Rainer Koschke's doctoral thesis.

Furthermore, we plan to investigate if the difference between each technique's outputs and the references, is roughly the same as the difference between two experts (i.e., interobserver variation).

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, Stephan Kurpjuweit, Kristina Lundqvist, Minna Mäkäräinen, and Klaus Schmid for their manual analysis of the three subject systems used in our comparison. Last but not least, we would like to thank our reviewers for their valuable feedbacks. In particular, we are grateful that Dr. Schwanke accepted to go out of anonymity to suggest both improvement to this work and promising future directions and to offer the possibility of direct discussions.

Notes

1. N.B. 20% corresponds to 1 function match in a 4-5-functions-ADT (median size of these sets).
2. This 70% match criterion should not be confused with the 20% threshold used in Section 5.3 to establish when two atomic components should be considered unrelated.
3. The results for Pure Internal Access presented here differ from those previously published. The previous approach did not consider internal access application of standard operators to base types. Furthermore, the previous implementation did not follow multiple levels of indirection while identifying internal accesses. As a result, some internal accesses were ignored.
4. The results for Delta IC presented here differ from those previously published. The previous implementation merged the produced atomic components which overlapped, in this article these atomic components are merged only if they overlap by more than 70%.

References

Aamodt, A. 1994. Explanation-based case-based reasoning. *Topic in Case-Based Reasoning*, Springer Verlag, pp. 274–288.

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

Canfora, G., Cimitile, A., and Munro, M. 1996. An improved algorithm for identifying objects in code. *Journal of Software Practice and Experience*, 26(1):25–48.

Canfora, G., Cimitile, A., Munro, M., and Taylor, C.J. 1993. Extracting abstract data type from C programs: A case study. In *International Conference on Software Maintenance*, pp. 200–209.

Dean, T.R. and Cordy, J. 1995. A syntactic theory of software architecture. *IEEE Transaction of Software Engineering*, 21(4):302–313.

Garlan, D. and Shaw, M. 1993. *An Introduction to Software Architecture Vol. 1, Advances in Software Engineering and Knowledge Engineering*, New Jersey, World Scientific Publishing Company.

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

Girard, J.-F. and Koschke, R. 1997. Finding components in a hierarchy of modules: a step towards architectural understanding. In *International Conference on Software Maintenance*. Bari, pp. 66–75.

Girard, J.-F., Koschke, R., and Schied, G. 1997. Comparison of abstract data type and abstract state encapsulation detection techniques for architectural understanding. In *Working Conference on Reverse Engineering*, Amsterdam, The Netherlands, pp. 66–75.

Guttag, J. 1977. Abstract data types and the development of data structures. *Communications of the ACM*, 20(6):396–404.

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

Liskov, B. and Zilles, S.N. 1974. Programming with abstract data types. *SIGPLAN Notice*, 9(4):50–60.

Liu, S. and Wilde, N. 1990. Identifying objects in a conventional procedural language: An example of data design recovery. In *Conference in Software Maintenance*. IEEE Computer Society Press, pp. 266–271.

Luckham, D.C., Kenney, J.H., Augustin, L.M., Vera, J., Bryan, D., and Mann, W. 1995. Specification and analysis of system architecture using rapide. *IEEE Transaction of Software Engineering*, 21(4):336–355.

Perry, D. and Wolf, A. 1992. Foundations for the study of software architecture. *acm SIGSOFT*, 17(4):40–52.

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

Schwanke and Hanson 1994. Using neural networks to modularize software. *Machine Learning*, 15:137–168.

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

Shannon, C.E. 1972. The *Mathematical Theory of Communication*. Urbana: Univ. of Ill. Press. ISBN 0-252-72548-4.

Shaw, M., DeLine, R., Klein, D.V., Ross, T.L., Young, D.M., and Zelesnik, G. 1995. Abstraction for software architecture and tools to support them. *IEEE Transaction of Software Engineering*, 21(4):314–335.

Sommerville, I. 1992. *Software Engineering*, 4th edition, Addison Wesley.

Yeh, A., Harris, D., and Reubenstein, H. 1995. Recovering abstract data types and object instances from a conventional procedural language. In Wills, L., Newcomb, P., and Chikofsky, E., editors, *Second Working Conference on Reverse Engineering*, Los Alamitos, California. IEEE Computer Society Press. pp. 227–236.