Comparison of Abstract Data Type and Abstract State Encapsulation Detection Techniques for Architectural Understanding

Jean-François Girard
Fraunhofer Institute for
Experimental Software Engineering
Sauerviesen 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

In the context of the authors’ research on architectural features recovery, abstract data type (ADT) and abstract state encapsulation (ASE, also called abstract object) have been identified as two of the smallest components which are useful to build a significant architectural overview of the system. The authors have named these the atomic components of an architecture.

This paper compares five published techniques which extract ADT and ASE from source code. A prototype tool implementing each technique has been developed and applied to three medium-size systems written in C (each over 30 Kloc). The results from each approach are compared with the atomic components identified by hand by a group of software engineers. These people did not know the automatic techniques which were going to be applied to the systems.

1. Introduction

The Bauhaus project aims at recovering multiple views of the software architecture of a system “as it is implemented”. That is, an overview would indicate the main components of the system, their connectors (how they communicate), and the constraints on these connectors and components. More complex components and connectors can be decomposed into simpler components and connectors. For example, a scanner and a parser are components of a compiler, whereas the scanner might have as a subcomponent an input buffer.

We have called atomic components the smallest components that are significant at the architectural level. While performing a case study [Gira97a], we have observed that abstract data types and abstract state encapsulations (also called abstract objects) are two instances of these atomic 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 or object instances.

In real systems, these atomic components often correspond to implementation of well-understood concepts which are a good starting point to look for more complex components. Such a starting point is crucial in architectural description recovery. In addition, ADT’s and ASE’s are important parts of a system even outside the architectural point of view. Their role has been recognized to provide information hiding and to support maintainability [Ghez91] and reuse [Somm92].

The question is, how to detect such atomic components when facing a language which does not have a mechanism to express them directly or when programmers do break the encapsulation and directly access the internal data. There is rich literature in reverse engineering on their detection. In this paper, five techniques have been selected and implemented, to be empirically studied. The results of each approach are compared with atomic components identified by software engineers, in order to identify the approach or combination of techniques which could help recover ADT and ASE closest to those identified by software engineers and which hence would be able to best support architecture recovery.

Paper Overview

The remainder of the article is organized as follows:

Section 2 gives a brief overview of each of the five
2.1. Routine Signatures and Refinements

This subsection presents three techniques of extracting ADT. Their common base is presented first, then the variations are explained.

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 [Somm92]. 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 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 no 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 which 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 - we even have to look at several types in a parameter list to form one abstract data type. For example, we 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.

In order to distinguish between operations of an abstract data type and routines that do not really belong to the abstract data type, several heuristics have been proposed:

- **Same Module heuristic**: It groups together only those routines, global variables, and types that are declared in the same module. We used this heuristic in [Gira97a].
- **Part Type heuristic**: It filters types in a parameter list that are part of another type in the list. Liu and Wilde proposed this heuristic in [Ogan94, Liu90].
- **Internal Access heuristic**: It considers only those routines as operations of an abstract data type that access components of that type. Yeh et al. presented this heuristic in [Yeh95].

**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 which are declared in the same module. In 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 assumes that programmers
are disciplined and follow this convention. 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.

**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 sorts of abstract data types we will have an operation that takes an element and puts it into the container. For the process queue, for example, there will be an insert routine with two arguments: the process that has to be inserted and the queue itself. We would not consider insert to be an operation for processes but for queues. The Part Type heuristic reflects that:

- a type PT which 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
- a routine R belongs to a type T if T is contained in R’s signature and there is no other type T’ in R’s signature with T < T’

As opposed to the Same Module heuristic, the Part Type heuristic is invariant against a programmer’s distribution of routines into modules. But it assumes that a part type is actually used to put it into its container or to retrieve it from it, respectively. Since it does not further analyze the actual usage, it is going to fail if the assumption is false. For example, if the part type relationship is only by accident, or if the routine just passes the two arguments to a real operation of the abstract data types, it will consider the routine to be an operation of the abstract data type.

**Internal Access Heuristic.** According to Sommerville, an abstract data type 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. From this idealized definition it follows 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 (Yeh et al. originally only proposed to consider internal access to record types, but the same can be applied to arrays and pointers as well):

- 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.

As opposed to the Part Type heuristic, the Internal Access heuristic really checks how the parameter type is used. However, in real programs we often find Sommerville’s rule broken. This often happens for reasons of efficiency or convenience in 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. However, even then all routines which access internal components of a type have to be changed when the representation of this type changes. This indicates a high coupling.

### 2.2. Global Variable References

Two of the techniques presented in the last subsection have been extended by their authors to identify ASE.

The Same Module heuristic produces ASE candidates by grouping global variables and routines which access them, only if they are declared in the same module. Yeh et al. add a similar strategy to their Internal Access heuristic to identify ASE. It groups global variables and all the routines which access them, regardless of where they are declared. We will refer to this strategy as **global access**.

An alternative to extend internal access heuristic to detect ASE is to keep only the routines which access the internal parts of a variable. We will refer to this strategy as **pure internal access**.

Each approach merges 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.

### 2.3. Interconnectivity Metric

The approach proposed by Canfora et al. [Canf93, Canf96] uses a heuristic to generate ASE candidates. These candidates are then ranked according to an index which measures the variation “internal connectivity” of the system due to the introduction of these ASE. 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 graph which 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 the set of all functions which set or use $v$

The candidates generated around function $f$ consist of:
- the context of $f$
- closely-related functions of $f$

The internal connectivity measure ($IC$) and improvement in internal connectivity ($\Delta IC$) are defined respectively as:

$$IC(F) = \frac{\text{closely related functions of } F}{\text{functions related to } F}$$

$$\Delta IC(F) = IC(F) - \sum_{v \text{ in context of } f} \left( \frac{|\{f' | \text{context of } f' = \{v\}\}|}{|\text{functions referencing } v|} \right)$$

The original approach presented in [Canf96] uses the following algorithm:

```
repeat
    build type-variable-reference graph
    create cluster candidates using a heuristic
    for each candidate
        compute improvement in cohesion metric ($\Delta IC$) introducing these candidates
        if improvement $\geq$ 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
```

In this paper, 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 which 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.

The current paper 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.

### 2.4. Clustering Based on Similarity

Schwanke proposed a clustering of routines into modules based on a similarity metric [Schw91]. His work was aimed at detecting subsystems and therefore, he only considered routines. This approach was extended to detect atomic components in [Gira97b]. It considers routines, types, global variables, and relationships among them. 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:

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

wherein $\text{Common}(A, B)$ reflects the number of common attributes of $A$ and $B$ and $\text{Distinct}(A, B)$ 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 $A$.

### Figure 1. Programming Language Entities and Relationships.

$\text{Linked}(A, B)$ 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 $\text{Linked}$ and $\text{Distinct}$ 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 paper it is set to 0.

The similarity between components to merge is computed in each iteration of the clustering algorithm as $Sim(A \cup B, X) = (Sim(A, X) + Sim(B, X)) / 2$.

### 2.5. Investigated Techniques

The previous sections introduced individual techniques of detecting ADT and ASE. Table 1 lists the combinations
of these techniques investigated in the experiment described in section 3. Note that not all approaches detect both ASE and ADT.

3. Experiment Setup

In order to evaluate the proposed techniques, they were applied to three medium-size programs and the results were 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 explains the experimental setup and the analysis method used.

3.1. Systems Studied

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

3.2. Human Analysts

Software engineers were given the task of identifying atomic components in each system. The systems were completely unknown to them. Table 3 summarizes their experience and how the task was divided among 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.

Identify the existing atomic components present in this system. These are ASE, ADT, or a combination of both.

- Here we gave the definitions of abstract data type and abstract state encapsulation that we already presented in the introduction (section 1) of this paper.
- The key difference of ADT and ASE is that an ADT is built around a type and an ASE around a set of simple global variables. This can be decided automatically, so do not waste time writing it down. Just identify the functions, variables, and types which belong together because they are cohesive and correspond to the idea of ASE and/or ADT.
- In practice, programmers sometimes break the encapsulation principle, therefore we widen the definition of abstract state encapsulations and abstract data types to clusters of types or variables respectively with their accessor routines. The internal representation of ADTs and ASEs can be public.
- NB not all functions, variables or types have to be put into ASE, ADT & ADT/ASE.
- In general, your experience and understanding has more value than rules, you are the last judge of what constitutes an ASE/ADT.

Figure 2. Guidelines for Human Analysts.

Table 4 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>
<thead>
<tr>
<th>System</th>
<th>#ADT</th>
<th>#ASE</th>
<th>#Hybrid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Aero</td>
<td>10</td>
<td>16</td>
<td>1</td>
</tr>
<tr>
<td>Bash</td>
<td>22</td>
<td>15</td>
<td>6</td>
</tr>
<tr>
<td>CVS</td>
<td>13</td>
<td>36</td>
<td>6</td>
</tr>
</tbody>
</table>
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 atomic components identified found no aberrant components. Furthermore, the fact that none of the software engineers knew the automatic to be applied to the systems, prevented bias toward a specific approach.

3.3. Comparison of Candidate and Reference Components

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

The following comparison defines an approximate matching to accommodate the fact that the distribution of functions, global variables, and types is sometimes subjective. That is, software engineers do sometimes come up with different ADT’s and ASE’s given the same program. We treat one component S to be part of another component T (denoted by S << T) if at least 70 percent of the elements of S are also in T.

The relationship between the candidates and reference components can be classified into six different cases:

- 1~1 is a relationship between a candidate C and a reference component R; where C << R and R << C.
- 1~n is a relationship between a candidate C and reference components R_i; R_i << C (i > 1). In other words, the candidate was not detailed enough.
- n~1 is a relationship where several candidates C_i together built one reference component R: C_i << R (i > 1). In this case the candidate was too detailed.
- 1<~1 is a relationship between a candidate C and a reference component R when C<<R, but not R<<C and there is no C’ (C’ ≠ C) such that C’<R.
- 1>~1 is a relationship between a candidate C and a reference component R when R<<C, but not C<<R and there is no R’ (R’ ≠ R) such that C<<R’.
- n~m is a relationship where candidates and reference components have no recognizable connection.

The best result is a 1~1 relationship, of course. But even 1~n, n~1, 1<~1, and 1>~1 relationships are useful. A software engineer, given these candidates, could easily combine or split them into atomic components. An n~m relationship does not provide useful information to the users of the approach.

These six cases can be classified into 3 categories according to their usefulness to a software engineer looking for atomic components:

- good (1~1) - they require a quick verification to identify the few elements which should be removed or added to the atomic component;
- ok (1~n, n~1, 1<~1, and 1>~1) - they require more attention to split, combine or refine a component.

This category can be further subdivided:
- too detailed (n~1, 1<~1) - need to be combined to produce candidate comparable to reference;
- too large (1~n, 1>~1) - need to be split to produce candidate comparable to reference.
- bad (n~m) - they are not close enough to the real components to guide the software engineer’s work. This corresponds to the false positives of an approach.

3.4. 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:

\[ \text{accuracy}(C, R) = \frac{|C \cap R|}{|C \cup R|} \]

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.

4. Results

This section compares the atomic components recovered by the approaches to those identified by software engineers, both at the quantitative and qualitative level. Note that the human intervention suggested in Internal Access and Delta-IC methods was not performed in order to make the comparison fair to the other techniques.

4.1. Quantitative Comparison

The recovered numbers of ADT and ASE components for each system by each method (see section 2.5) are reported in Tables 5, 6 and 7. The recovered components are classified according to the categories: good, too detailed, and too large. Each number in the shaded area under the good and ok column is the absolute number (#) of recovered components and the average accuracy (acc). As some techniques produce only ADT or ASE, they are not presented in table where they do not apply.
These results can be summarized in Table 8. The entries of this table are computed as the accuracy multiplied by the number of components summed up for the good, too large, and too detailed categories. The best results are shaded in this table.

Table 8. Overall Comparison.

<table>
<thead>
<tr>
<th>Method</th>
<th>System</th>
<th>ADT</th>
<th>ASE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td># ac.</td>
<td></td>
</tr>
<tr>
<td>Same Module</td>
<td>Aero</td>
<td>6</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>Bash</td>
<td>5</td>
<td>5</td>
</tr>
<tr>
<td></td>
<td>CVS</td>
<td>23</td>
<td>4</td>
</tr>
<tr>
<td>Pure Internal Access</td>
<td>Aero</td>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Bash</td>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CVS</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Global &amp; Internal Access</td>
<td>Aero</td>
<td>1</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Bash</td>
<td>4</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>CVS</td>
<td>8</td>
<td>0</td>
</tr>
<tr>
<td>Delta IC</td>
<td>Aero</td>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td></td>
<td>Bash</td>
<td>5</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>CVS</td>
<td>6</td>
<td>5</td>
</tr>
<tr>
<td>Schwankel Similarity</td>
<td>Aero</td>
<td>3</td>
<td>0</td>
</tr>
<tr>
<td></td>
<td>Bash</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td></td>
<td>CVS</td>
<td>4</td>
<td>0</td>
</tr>
</tbody>
</table>

These results indicate that Same Module and Part Type are overall better than the other techniques, but the false positives of each approach have to be considered in the comparison.

False Positives

The accuracy is not defined for the bad (n ~ m) category. Table 9 shows the number of components wrongly identi-
fied by each technique, i.e., the false positives. It is interesting to note that Same Module and Part Type are both the techniques with the best results and with the lowest number of false positives.

Table 9. Number of False Positives.

<table>
<thead>
<tr>
<th>Method</th>
<th>aero</th>
<th>bash</th>
<th>cvs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Same Module</td>
<td>1</td>
<td>7</td>
<td>1</td>
</tr>
<tr>
<td>Pure Internal Access</td>
<td>6</td>
<td>54</td>
<td>21</td>
</tr>
<tr>
<td>Global &amp; Internal Access</td>
<td>6</td>
<td>54</td>
<td>21</td>
</tr>
<tr>
<td>Part Type</td>
<td>2</td>
<td>15</td>
<td>9</td>
</tr>
<tr>
<td>Delta IC</td>
<td>9</td>
<td>43</td>
<td>0</td>
</tr>
<tr>
<td>Schwank Similarity</td>
<td>55</td>
<td>134</td>
<td>46</td>
</tr>
</tbody>
</table>

Complementarity of Techniques

Before deciding to select one approach and exclude the others, one has to consider the additional information provided by the other techniques. As a first estimate of the contribution of each approach, we considered the good candidates from each technique. For each component identified by our software engineers and for each approach, we looked at the closest single candidates with an accuracy of more than 60%. Then we looked at how many new correct atomic components each approach found which were not identified by the Same Module heuristic. This information is presented in Table 10.

Table 10. New Correct Atomic Components.

<table>
<thead>
<tr>
<th>Method</th>
<th>aero</th>
<th>bash</th>
<th>cvs</th>
</tr>
</thead>
<tbody>
<tr>
<td>Identified by Software Engineers</td>
<td>27</td>
<td>43</td>
<td>54</td>
</tr>
<tr>
<td>Found by any approach</td>
<td>9</td>
<td>15</td>
<td>35</td>
</tr>
<tr>
<td>Found by more than 1 approach</td>
<td>3</td>
<td>4</td>
<td>13</td>
</tr>
<tr>
<td>Found by Same Module</td>
<td>7</td>
<td>7</td>
<td>34</td>
</tr>
<tr>
<td>Found by Part Type alone</td>
<td>0</td>
<td>6</td>
<td>0</td>
</tr>
<tr>
<td>Found by Delta IC alone</td>
<td>0</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>Found by Global &amp; Internal Access alone</td>
<td>1</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Found by Pure Internal Access alone</td>
<td>1</td>
<td>1</td>
<td>0</td>
</tr>
</tbody>
</table>

These data suggest that few new candidates that could not be found by Same Module alone are found by other techniques, except for bash, where Part Type found many.

4.2. Qualitative Comparison

After the quantitative comparison described in the last section we analyzed divergences between candidate and reference components for the respective methods.

Same Module. Same Module groups all connected routines, variables, and types in the same module. The postulate is that the programmer structures files according to atomic components. Detection of abstract data types with this heuristic did not work well for all systems. Often, there was a header file with system-wide type declarations. The routines, however, were implemented in several other C files that included the type declarations. Detection of abstract state encapsulation succeeded better, since global variables are never declared in header files (they can only be declared there as external).

Large files can be a problem for the Same Module heuristic. In CVS, for example, we found a type RCS node which encapsulates dependencies on the underlying revision control system (RCS) of CVS. This node type was declared in one huge file where many routines used it as a parameter. Consequently, Same Module created a very large atomic component candidate. The group of software engineers has refined this candidate into different aspects of the RCS subsystem.

Part Type. The Part Type heuristic is a refinement for detecting abstract data types to overcome erroneously large components. It attaches a routine to those types of its signature that do not have a containing type that is listed in the routine’s signature.

The assumption is that the part type is put into the container or retrieved from it. Compared to the other techniques, Part Type revealed most abstract data types with the highest accuracy rate. However, it still detected only a fraction of all reference components. Moreover, it does not check how the parameters are actually used, and therefore many routines which do not directly use the value of a parameter, but only pass this parameter to a routine they call (probably the true operator of the ADT) would be included into ADT. In case of CVS this could be observed for the RCS node type and the hash table type already mentioned.

Internal Access. The Internal Access heuristic groups user-defined data types and global variables with the routines that access their internal parts. This analysis goes beyond the Part Type heuristic in that it really checks how routines treat types and variables. This heuristic did badly in the comparison for at least two reasons. First, if there is a global table, such as an array of error messages, all readers of this table are considered operators of this abstract state encapsulation which yields an erroneously large component. Second, it does not work for primitive types, such as a file descriptor in Unix which is represented as an integer type. It is a common phenomenon for abstract state encapsulation to have separate global variables that together form an object. For example, stacks are often
declared as two distinct variables, one for the contents and one for the stack pointer. The latter is declared as integer and used as an index to the array implementation for the contents. Then the stack pointer will not be recognized as part of the abstract state encapsulations, since internal access to a primitive data type is meaningless. Since abstract state encapsulations have exactly one instance (otherwise they would be abstract data types), programmers do not make the effort to put them together in a record which would make their connection more obvious. But even if the routines are not considered operators of the atomic component, this analysis still recovers useful information. All these routines might have to be updated when the representation of the type changes.

Schwanke Similarity. Schwanke similarity iteratively merges components with maximum similarity until the maximum similarity value falls short of a given threshold. After merging two components we must compute the similarities of the new component to all other components. We investigated two different variants. Using $Sim(A \cup B, X) = \max(Sim(A,X), Sim(B,X))$ typically produced one huge component containing most of the system and a large number of very small components. Useful cluster sizes could be achieved using the average instead of the maximum, i.e. $Sim(A \cup B, X) = (Sim(A,X) + Sim(B,X)) / 2$.

The similarities between the initial components and the clustering process itself depend on several parameters. One fundamental problem was to fix these parameters in a sensible way. We did not succeed in finding (by trial and error) a parameter combination that produces good results for any of the three systems though we ran dozens of analyses with various parameters. Finally, we used threshold (aero & bash = 0.1, cvs = 0.2) for the clustering loop. The factors k=1.0 and d=3.0 turned out to be appropriate.

Delta-IC. Delta-IC-based clustering uses a heuristic to generate candidates around each function. For a function F this candidate is composed of all the variables set or used by F and all the functions which only access those variables. Once these candidates are generated, the approach computes the improvement in an internal connectivity metric (delta-ic) due to the candidate introduction. Every candidate with a threshold above delta-ic is kept.

Due to the comparison between techniques, we could not slice functions which belonged to more than one candidate. As a result, we decided to merge overlapping candidates before comparing them to the reference components in order to have non-overlapping candidates for comparison.

The major difficulty is establishing the right thresholds. They depend on the specificity of the system studied and cannot be directly reused (bash & aero = 0.18 cvs = 0.51). One practical solution is to take a sample of the system and perform manual recovery by applying the approach on the sample and adjusting the threshold accordingly. It is necessary to compromise between the efforts required to analyze a sample manually and the quality of the results. With the mentioned threshold, relatively interesting results were obtained.

It is a topic of further research to establish the relationship between sample size and result quality.

Factors to Consider. The software engineers were not always sure whether a routine belongs to one atomic component or to another. Examples for such debatable cases are transformation routines from one atomic component into another. It is not always clear whether they belong to the target type of the transformation, if there is a high coupling to the source type as well. Another problem are closely-related types. For example, in CVS there is a hash table implementation that contains two intertwined types, one for lists and one for nodes. The one for nodes has a pointer that denotes the next element in the list. Nevertheless, the group of software engineers considered them as two distinct abstract data types. That is, the concepts of abstract data type and state encapsulation are vague, and therefore no approach will ever detect atomic components with absolute accuracy. However, they are the basic components of software architectures. They have to be understood to understand the whole system.

5. Conclusions

A previous case study lead the authors to regard abstract data types and abstract state encapsulations as atomic components which are useful in producing an architectural view of a system. In this paper we compared five existing techniques to detect atomic components by applying them to three publicly available medium-size systems written in C (bash, cvs, aero). Two groups of software engineers compiled a list of reference atomic components for each system. These lists were used to compare the results of the techniques with human judgement.

According to this comparison, the Same Module heuristic identifies more abstract state encapsulations and hybrid atomic components and produces less false positives than any other approach. However, it relies on the assumption that programmers usually declare distinct atomic components in distinct modules. This assumption might be too strong for systems other than those studied, especially for legacy systems.

For the identification of abstract data types (ADT), the Part Type heuristic performed slightly better than the other techniques. In the case of bash, it showed that it can complement the Same Module approach well. However, it can only recover ADT’s.
There are three alternatives to replace Same Module heuristic when its assumption does not hold: Delta-IC, Schwanne Similarity and Internal Access.

Both Schwanne Similarity and Delta-IC have parameters which depend on the system studied and have to be adjusted. This calibration requires time, and finding the ideal parameter or a close approximation quickly is not likely. Schwanne Similarity did not perform well in this study. Delta-IC performed well for bash and cvs, but it can only extract ASE. Internal Access seems to offer a better compromise, it has reasonable results and does not require the adjustment of parameters.

This study hinted at the level of support that tools could provide in identifying atomic components. The techniques can quickly yield candidates that can be checked by software engineers within minutes. However, all techniques could only recover a small portion of the atomic components that were listed by software engineers. And many candidates they provide correspond only roughly to those atomic components; i.e., elements of these atomic components were superfluous or lacking.

Whereas the groups of software engineers needed about 20 hours to compile the list of atomic components for each of our subject systems, each atomic component produced by the techniques can be checked by software engineers within minutes.

Future Work

This experiment will be extended to a larger number of systems to obtain more significant results and to be able to evaluate how far these conclusions can be generalized. Studying the behavior of automatic techniques to recover atomic components on a set of systems where they are identified by humans can help us see possibilities for improving these techniques.

Combinations of techniques will be explored to try to obtain more complete results and filter the false positives.

However, it is already clear that new techniques which could identify more atomic components are needed to obtain most of the atomic components which are useful to identify the more complex components of the architecture of a system.

Acknowledgment

We would like to thank Jean-Marc DeBaud and Erhard Plödereder for their support of this project and their insightful inputs to this research. We thank the reviewers for their valuable suggestions and remarks. 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


