Copyright 1999, IEEE. Published in the Proceedings of the Working Conference on Reverse Engineering, WCRE'99, October 5-7, 1999 in Atlanta, Georgia, USA. Personal use of this material is permitted. However, permission to reprint/republish this material for advertising or promotional purpose or for creating new collective works for resale or redistribution to servers or lists, or to reuse any copyrighted component of this work in other works, must be obtained from the IEEE. Contact: Manager, Copyrights and Permissions / IEEE Service Center / 445 Hoes Lane / P.O. Box 1331 / Piscataway, NJ 088-1331, USA.

An Incremental Semi-Automatic Method for Component Recovery

Rainer Koschke

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

Abstract

Atomic components are sets of related variables, types, and subprograms, e.g., abstract data types and objects. Many techniques exist to detect them automatically. However, as an evaluation has shown, none of them has the precision needed [9]. One approach to achieve a higher precision is to integrate the user into the detection cycle. This paper describes a method in which computer and human work together to find atomic components. Furthermore, it discusses how the techniques can be enhanced to work incrementally, which is needed if they are to be integrated with this method. Moreover, it proposes ways of combining the techniques within this interactive method.

1. Introduction

Architecture recovery comprises detection of **components** (the computational parts) and **connectors** (the means and points of communication) of systems. The most primitive components consist of subprograms, types, and global variables. Groupings of these kinds of declarations are, for example, objects, abstract data types, and sets of related routines, and are called **atomic components** in the following since they do not consist of smaller components. Atomic components can be viewed as cohesive logical modules. Livadas and Johnson [15] give several reasons for recovering atomic components: Support for understanding system design, testing and debugging, reengineering from a procedural programming language to an object-oriented language, avoidance of degradation of the original design during maintenance, and facilitation for reuse.

The goal of our research is to find techniques and methods for atomic component detection in the general framework of architecture recovery. In an experiment, we have evaluated several published approaches to detect abstract data types and objects [9]. The overall result was that none of the techniques has the precision needed. There are several alternative approaches to overcome this: The techniques can be combined, other sources of information can be considered (for example, dataflow information or domain knowledge), or the user could be integrated into the search. This paper describes a method in which computer and maintainer work hand in hand to detect atomic components. Within this interactive framework, the techniques can be combined by simple operations triggered by the user. Due to the complexity, vagueness, and to some degree subjectivity, it seems questionable whether we can ever find precise techniques that fit all cases. Therefore, atomic component recovery is a problem that has to be tackled in concert with a maintainer at any rate. Hence, how this can be effectively achieved should be investigated first before we search for other sources of information.

Paper outline

The paper first briefly summarizes available automatic base techniques for atomic component detection (Section 3). Then it describes a method in which human and computer interact to detect atomic components that integrates the basic techniques (Section 4). In Sections 5 and 6, it will be discussed how the base techniques can be combined in order to support the method. Then a strategy is suggested for atomic component detection within this framework (Section 7). Section 8 compares the new semi-automatic approach to other existing interactive approaches. We begin with the terminology used in this paper.

2. Terminology

In this section, terms are defined that are used throughout of this paper.

An **entity** is a global programming unit with a name. A **base entity** is a variable, a type, or a subprogram. An **atomic component,** or short **component,** is a set of base entities that form a concept relevant at the architectural level. Examples are abstract data types and objects. A **candidate atomic component**, or short **candidate**, is an atomic component proposed by a technique and not yet confirmed by the user. The **elements of an atomic component A** are its constituting variables, types, and subprograms, denoted by *elements(A)*. A **view** is an excerpt of the current knowledge about the entities in the system and their relationships. The **base view** consists of all base entities and their relationships extracted from source code. An **atomic component view**, or short **component view**, describes the decomposition of the atomic components into its constituents. The **user view** is an atomic component view whose contents are confirmed by the user. A base entity is consid-

ered **bound** when it is a constituent of an atomic component in the user view; otherwise it is considered **free**.

3. Automatic Base Techniques

There are many existing automatic techniques for atomic component detection in the literature. They can roughly be classified as follows according to the kind of information they leverage:

- domain-model-based approaches use a domain model that describes the domain concepts and their relationships and that is used to guide the search for components [6]
- dataflow-based approaches leverage dataflow information [23]
- structure-based approaches are based on base entities and their structural relationships; structure-based approaches can further be distinguished into connection-, metric-, graph-, and concept-based approaches (Canfora et al. independently came to a similar classification [3]).

In the following, only structure-based techniques will be discussed in more detail because these are the techniques integrated into the current prototype supporting the semi-automatic method. There is no principal reason why the other two kinds of approaches could not be integrated as well. The only requirement for a technique to be integrated into the framework is to present its results as component view.

Table 1. Connection-based approaches

	Connected_Entities	Reference
Same Module	signature types, i.e., parameter or return types of a subprogram, and accessed variables declared in the same module	[7]
Part Type	signature types that are not a part-type of another type in the same signature; T is a part-type of T' if T is (transitively) used in the declaration for T'	[18]
Internal Access	signature types and variables whose record components are accessed	[25]
Same Expression	accessed variables that occur in the same expression	[13]

Connection-based approaches cluster entities based on a specific set of direct relationships between entities to be grouped. One can unify these techniques by defining a function **Connected_Entities** for each one that returns the base entities to which a given subprogram should be grouped as described by Table 1.

Metric-based approaches cluster entities based on a metric using an iterative clustering approach. Schwanke's [20] and our *Similarity Clustering* approach [8] and *Type-based Cohesion* [17] fall in this category. Schwanke's Similarity Clustering is aimed at finding related subprograms based on direct call relationships among the subprograms and common and distinct usages of non-local names. Our *Similarity Clustering* approach distinguishes among different kinds of usages of non-local entities and adds informal information. *Type-based Cohesion* groups subprograms according to the portion of types of their parameters, local variables, and the non-local variables they reference where each occurrence of a type counts (as opposed to the other approaches that count each non-local entity only once).

Belady and Evangelisti's approach groups related subprograms using a similarity metric based on data bindings [1]. A data binding is a potential data exchange via a global variable. Hutchens and Basili extend Belady and Evangelisti's work by using a hierarchical clustering technique to identify related subprograms and subsystems [11].

Delta-IC is actually a hybrid of connection-based and metric-based approaches since it consists of two parts: cluster formation and cluster filtering [2]. The actual *cluster* is built around a subprogram *S* and comprises the *closely related subprograms* of *S*, i.e., all subprograms that access only variables accessed by S, and the referenced variables of *S.* The metric measures references from outside the cluster (coupling) and occurrences of minimal subclusters consisting of subprograms that reference exactly one of the variables in the cluster (cohesion).

The metric-based approaches differ from connection-based approaches by the degree of freedom that is offered by the metrics parameters and the threshold that can be varied to find atomic components with varying reliability.

Graph-based approaches derive clusters from a graph by means of graph-theoretic analyses. The difference to connection-based approaches is that the whole graph has to be considered whereas connection-based approaches regard only direct relationships between entities in order to decide whether they should be grouped. *Strongly Connected Components Analysis* considers cycles in the call graph as components and *Dominance Analysis* identifies local utility functions of atomic components [4].

Concept-based approaches use concept analysis to compute a lattice of concepts. A concept is a maximal set of objects sharing common attributes where each object has all attributes [14]. In order to find atomic components with concept analysis, a subprogram is considered an object in the sense of concept analysis. Attributes could be: accessing a certain variable, having a certain signature type, and so forth. In the ideal case, the concept lattice consists of

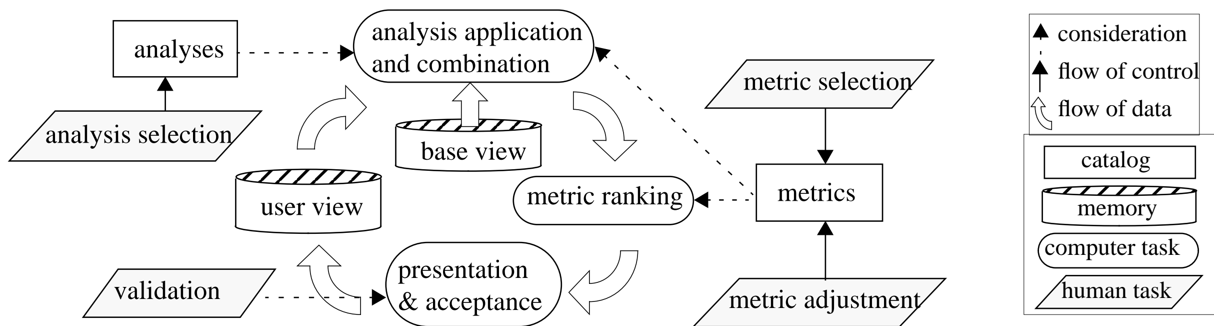

Figure 1. Semi-automatic method for component recovery

separate sublattices that are only connected to the top and bottom element of the lattice. Such sublattices are component candidates [14]. In practice, however, due to violations of the information hiding principle, the lattice of concepts has many interferences. Several heuristics to detect atomic components within this lattice despite of interferences have been proposed [21, 3, 19]. Another drawback of this approach is that it may take exponential time to find a concept lattice in the worst case. Practical experiences have indicated that it still takes cubic time on average [22].

Research in atomic component detection has mostly concentrated on the automatic techniques. This has lead to many interesting, yet isolated techniques. A notable exception are Canfora et al. who have proposed to combine concept analysis with other heuristics in order to simplify the concept lattice [3]. However, the other heuristics are only subordinated to concept analysis in their approach. Moreover, little attention has been paid to the question how a maintainer could be integrated into the detection process.

Contribution. This paper contributes to component detection by describing a semi-automatic method that incorporates existing techniques, shows how these techniques can be combined and applied systematically, and how the user can be integrated. Furthermore, it explains how structure-based techniques can be extended to work incrementally.

4. The Semi-Automatic Method

The proposed semi-automatic method integrates the maintainer into the detection process right from the beginning. The automatic techniques are used to yield component candidates that are validated by the user.

Figure 1 contains the main constituents of the method. The inner cycle, consisting of analysis application, metric ranking, presentation, and bookkeeping of detected components, is the core of the detection process. The detection process is controlled by selected analyses, metrics, and user validation.

The base view contains the base entities and their relationships needed for component detection and is automatically derived from source code. The role of the user view is to record the components that have been detected and validated by the user so far. In the beginning, when no component is known, it is empty. The user selects an analysis that is to be applied. The analysis takes into consideration the components that were previously confirmed by the user (in the first iteration there are none). Thus, the analyses are applied incrementally. Generally, the techniques propose many candidates. The user should not be swamped with all of them. Instead, the candidates should be presented in their presumed quality. They can be ranked by certain metrics selected by the user. Many of these metrics come with parameters that can be adjusted by the user. The metrics of the metric-based techniques can be used for clustering as well as for ranking (that explains the arrow from *metrics* to *analysis application* in Figure 1). After the candidates have been ranked, the candidates are presented to the user for acceptance. The presentation is a crucial and non-trivial task. It must be in such a way that the user's validation can be as quick as possible. Additional information the user may need has to be provided on demand. For example, the maintainer will probably also want to inspect the source code. The user validates the candidates and those component he or she accepts enter the user view.

In each iteration, the user selects and combines different analyses to find components that could not be found by previous analyses. The process ends when the found components are sufficient for the task at hand or no further component can be found anymore. Section 7 will recommend a strategy for the selection of analyses.

The user does not have to select, apply, and validate one analysis at a time. Instead, several analyses can be selected and applied in parallel. Then, the intersection, union, and

differences of these analyses can be automatically ascertained, and the user can investigate and validate these. Particularly large candidates of some techniques can be refined by applying other techniques to these individually.

Because the typical maintenance task does usually not require to find all components of a system but only a few relevant ones in a specific part of the system, the domain of search can be restricted to certain modules.

The following sections go into more detail of the individual steps of the method.

4.1. Analyses

Currently, the framework offers all connection-based, metric-based, and graph-based techniques listed in Section 3.

As Figure 1 indicates, a technique uses the base view and the user view as input. The analysis may add free entities to the existing components in the user view or may propose new candidates. Thus, the techniques are applied incrementally. The original techniques as published are not incremental. Section 5 will, therefore, describe how they can be extended. Since it does not make a difference for incremental techniques whether the input component view is the user view or a component view generated by another technique, the incremental techniques basically represent functional composition. Functional composition is one way of combining techniques. Large components of one technique can individually be refined by other techniques and base entities left by one technique can be grouped by the next technique.

Another way of combining the techniques is to apply set operators, namely, union and intersection, to their results since the techniques are basically functions that yield sets of components. However, this is not so simple as Section 6 will describe.

Providing operators for combining the results of techniques was preferred to a technical combination of the techniques themselves. It eases integration of new techniques and gives the user the choice of combinations.

4.2. Metric Selection, Adjustment, and Ranking

Metrics are used to assess and rank the candidates that have been proposed by the analysis. There is a catalog of metrics that the user can choose of. The catalog comprises the metrics of the metric-based approaches. Furthermore, the underlying heuristics of the other techniques can equally be expressed as metrics and used for ranking [13]. Established intra-modular and inter-modular metrics, such as number of lines of code, McCabe or Shepperd complexity, can be used to measure additional aspects of the candidates [5]. The metric used for ranking is a composite metric that is the normalized weighted sum over the individual metrics.

The composite metric is used to guide the user through the large set of candidates. The metric is computed once and then a threshold is used to control the presentation. All candidates above the threshold come to the fore. The user can start with a high threshold that is decreased step by step. In each step, the candidates above the chosen threshold are validated. Accepted and rejected parts of the candidates are not presented again in the following steps.

Some metrics have parameters that need to be adjusted. Altogether, there are hence three dimensions of variability: The weights of the basic metrics within the composite metric, the inherent parameters of the basic metrics, and the filtering threshold. All these parameters can be adjusted by the user and the presentation updated accordingly. Several distinct metric settings can be tried without need to rerun the analysis.

4.3. Presentation, Validation, and Acceptance

For presentation and interaction with the user, the customizable graph editor for reverse engineering, Rigi, is used [16]. Rigi offers many useful capabilities such as:

- support for annotated nodes and edges of different types
- hierarchical nodes and views
- direct linkage to the corresponding source code by clicking on nodes
- automatic layout and context-preserving browsing capabilities
- filter and selection mechanisms
- Rigi command language for customizations

We extended Rigi in many directions to adapt it to our needs. The adaptations were opportunistic; not everything that might have been useful could be worked into Rigi, e.g., an undo mechanism. But all of our major requirements were more or less easy to fulfill with Rigi.

In order to distinguish the original Rigi from Rigi with our extensions, we will refer to the former as **original Rigi** and to the latter as **extended Rigi** in the following.

The analyses can be selected, combined, and started from within the extended Rigi by means of list boxes and menus. It was important to us that the selection and combination is easy to do with simple mouse clicks such that the user need not learn a complex language.

The result of an analysis is represented by a single hierarchical *analysis node* containing the actual candidates. This makes it possible to further process the results of an analysis by applying commands from a context-sensitive mouse menu to this node. For example, the difference to

the currently accepted atomic components can be shown, it can be intersected or united with the results of another analysis, or the next kind of analysis can be applied to it.

The analysis node can be unfolded. Then, the actual atomic component candidates are shown. The user can browse these candidates by clicking on the nodes or viewing the node hierarchy as a whole. The node hierarchy is especially interesting when the results of *Similarity Clustering* or *Type-Based Cohesion* are viewed. These two clustering approaches return a tree that indicates the order in which elements were grouped together and so immediately show what is more similar and what is less. The maintainer can then “climb up the tree” starting at the leaves and stop at one inner node for which the combination is doubtful. Direct validation is possible in any view.

The returned candidates can be accepted or rejected individually or as a whole by direct manipulation. The atomic components can be renamed by the user to give them a meaningful name. Single base entities within candidates can be accepted or rejected. Cut and paste capabilities are available to move single or whole sets of base entities as a group from one atomic component to the other. Any base entity can be added to a candidate. Maintainers are also able to create their own atomic components.

Everything confirmed by the user is moved to the user view. The user view is likewise represented as analysis node such that most commands available for analysis results are also available for the user view in a uniform manner. Only those that make no sense for the user view were excluded, such as accepting nodes or viewing the difference to the user view.

The user can add positive and negative information. **Positive information** expresses that a base entity belongs to a given atomic component; this is added when the user confirms an atomic component. **Negative information** conveys that two entities do not belong together, i.e., they exclude each other mutually.

More precisely, the symmetric relationship **mutually-exclusive**(a, b) among two entities a and b and added by the user expresses that a and b must not be added to the same component. Likewise, if *mutually-exclusive*(a,b) and a (or b) is a component, b (or a) must not be added to a (or b). If *mutually-exclusive*(a, b) and b is a part of component c, *mutually-exclusive*(a, c) is always induced.

Every analysis must preserve all positive information, that is to say, an analysis may only add to the atomic components that a user has confirmed and never remove any of their elements, and likewise, an analysis must not cluster entities that were not supposed to be grouped together.

5. Incremental Base Analyses

The techniques have originally not been proposed as incremental techniques. The needed enhancements of the basic techniques to work incrementally are briefly described in this section. A more detailed description can be found in [13].

An incremental technique has as input a description of the system that does not only contain the base entities and their relationships but also a set of atomic components already detected by the first technique plus a description on the mutually exclusive entities. The composition, i.e., all incremental analyses, may only group such base entities in the input base view that do not already belong to atomic components in the input component view. We will call base entities **bound** if they are already part of a component in the input component view. All other entities are considered **free**.

The composition can be organized in the following steps (the term *cluster* is used here to make clear that the individual techniques generate sets of related elements that only become components in the last stage of the composition):

1. Iterate over the free entities and cluster them.
2. Split all clusters so that there are no mutually exclusive entities in the same subcluster.
3. Transform clusters into components.

The first step depends on the analysis. It is described as composition in the following sections. The second and third steps are identical for all analyses and are explained in Section 5.4 and Section 5.5. An advantage of organizing the composition this way instead of letting the analyses be in charge of mutually exclusive entities is that the analyses do not have to take care of negative information.

We will discuss the diverse classes of approaches, namely, connection-based, metric-based, and graph-based techniques separately.

5.1. Incremental Connection-based Techniques

Connection-based approaches iterate over the free entities in the base view and collect the connected entities that should be grouped (Figure 2). The connected entities may be either bound or free. Free connected entities will belong to the same cluster of the entity that is under consideration. Bound connected entities cannot be grouped again because they already belong to a component. Instead, the entity under consideration should belong to the same component the connected bound entity belongs to. This will technically be solved by adding the enclosing component in lieu of the connected entity. The resulting clusters will be transformed into atomic components as described in Section 5.5.

The algorithm shown in Figure 2 implements this strat-

egy. It uses the union-find data structure for disjoint sets described by Hopcraft and Ullman [10] and assumes that all entities are enumerated from 1 to *Last_Entity*.

Generic Parameter:

- Connected_Entities : Entity → Set of Entities

Input:

- base view B
- component view A

Output:

- set of disjoint clusters

Algorithm

```
-- Put each free entity and component in a set of its own:
for E in 1..Last_Entity
  where not Is_Bound (E, A) or Component (E) loop
    New_Set (E);
end loop;
-- Iterate over free entities and cluster connected entities:
for E in 1..Last_Entity
  where not Is_Bound (E, A) and not Component (E) loop
    for C in Connected_Entities (E) loop
      if not Is_Bound (C, A) then
        Union (Find (E), Find (C));
      else
        -- add enclosing components of C in lieu of C
        for AC in Enclosing_Components (C, A) loop
          Union (Find (E), Find (AC));
        end loop;
      end if;
    end loop;
  end loop;
Result:
  Each disjoint set represents a cluster that constitutes a candidate.
```

Figure 2. Incremental connection-based technique.

In the example scenario in Figure 3, there are two existing components AC_1 and AC_2 where *elements* $(AC_1) = \{T_1, V_1\}$ and *elements* $(AC_2) = \{T_2\}$. T_1 and T_3 as well as F_2 and F_3 are mutually exclusive. *Connected_Entities* $(F_1) = \{T_1, T_3\}$, *Connected_Entities*$(F_2) = \{V_1\}$, and *Connected-Entities*$(F_3) = \{V_1, T_2\}$. Due to the overlap among these sets, the algorithm described in Figure 2 produces the cluster $\{F_1, F_2, F_3, T_3, AC_1, AC_2\}$ where T_1, V_1, and T_2 have been replaced by their respective enclosing component.

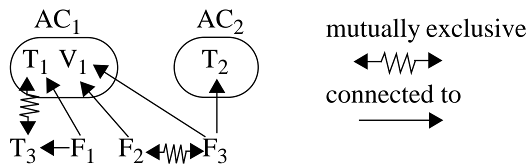

Figure 3. Example for incremental application.

5.2. Incremental Metric-based Techniques

A single step in the incremental connection-based techniques basically consists of two parts: (1) select a cluster and (2) replace bound entities within these clusters by their enclosing components. An incremental extension of *Delta IC* can be organized analogously:

1. Initially, the clusters are identified according to the pattern that is used by *Delta IC,*
2. then clusters whose *Delta IC* value is below the threshold are filtered,
3. and, eventually, bound entities are replaced within these clusters by their enclosing component.

The generic algorithm in Figure 2 can be used to implement the incremental version of *Delta IC* by defining a function that implements step 1 and 2 used as *Connected_Entities* for the instantiation.

Both *Type-based Cohesion*, *Similarity Clustering,* and *Data Binding Clustering* use the same hierarchical clustering algorithm; they only differ in the underlying metric. In an incremental approach, components have already been detected and yet free entities have to be clustered. This compares to a snapshot of the clustering algorithm after a few runs when there are already some clusters and yet more iterations ahead. Considering this, the clustering algorithm can easily be modified to work incrementally. Only a pre- and a post-processing phase is necessary. In the pre-processing phase, the similarity relation is computed among all components and all free entities using a group similarity [8]. If there are nodes that are mutually exclusive, their similarity is set to 0. The clustering algorithm then clusters all components and free entities based on the similarity relation just computed. The results are clusters that may contain components and base entities. These clusters are then treated in a post-processing phase as described in Section 5.5.

5.3. Incremental Graph-based Techniques

Graph-based approaches derive clusters from a graph by means of graph-theoretic analyses where the whole graph is considered. In the beginning, when no components are known, this graph is the base view. When applied incrementally, however, components have been partially recognized. Hence, it is known that certain nodes form a unit and, therefore, the graph analyses should be applied to the graph in which related nodes have been grouped together, i.e., to the graph that results from the following two transformations:

- all bound entities are replaced by their enclosing component
- all former connections of the bound entities are redi-

rected to the new component node

This will be called the **collapsed graph**. Then, the candidates of the incremental *Strongly Connected Component Analysis* are the cycles in the collapsed graph. Likewise, dominance analysis can be applied to the collapsed graph in order to identify local entities of a component. In the case of *Dominance Analysis*, no new clusters are proposed; instead, a component absorbs all entities it dominates.

5.4. Handling Mutually Exclusives in Candidates

Being in a common cluster proposed by an analysis means for the entities of a cluster that the analysis "believes" they belong together. However, if some of these entities are mutually exclusive because the user has previously disagreed with this grouping (negative information), they must not be put into the same candidate. Nevertheless, there may be entities in the cluster that are not in conflict and, therefore, the cluster is not completely meaningless. Instead of throwing away the whole cluster, the cluster should be partitioned into subclusters without mutually exclusive entities. A second requirement for a reasonable splitting is that the subclusters should be as large as possible. Subclusters with only one element obviously do not have any conflicts, but they are not very helpful either. Unfortunately, we are facing the NP-complete graph coloring problem here, i.e., for an optimal solution, we may need exponential time.

The graph coloring problem is to assign a minimal number of colors to nodes in a graph where no two neighboring nodes may have the same color. This is equivalent to partitioning the nodes of a graph into subsets where no two neighboring nodes are in the same set. The same problem exists for register allocation in compilers. The usual way there in tackling this problem is to use a heuristic in which the nodes are removed from the graph in the ascending order of their number of remaining conflicts and put onto a stack. When all nodes are on the stack, the nodes of the stack are popped and assigned to a partition such that no neighboring nodes are in the same partition. The partition assigned to a given node *N* is the minimal available partition that does not contain a node that is in conflict with *N*.

Proceeding exactly on the strategy just described may result in subclusters of unconnected entities. Subclusters of unconnected entities can be avoided by two additional heuristics that can easily be integrated with the scheme above: Firstly, subprograms are added to the stack before types, variables, and components such that the latter are partitioned first and can be used as crystallization points and secondly, an entity *E* is added to the partition that contains most connected entities of *E*. If there is no partition with a connected entity, *E* is added to a new partition.

For the example cluster $\{F_1, F_2, F_3, T_3, AC_1, AC_2\}$ generated by an incremental connection-based technique for the scenario in Figure 3 (see Section 5.1), we obtain the interference graph in Figure 4 (the components AC_1 and AC_2 inherit the relationship of their elements).

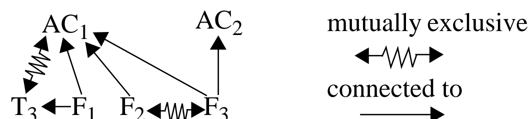

Figure 4. Resulting interference graph.

The entities may be pushed onto the stack in the order F_1, AC_2, F_2, F_3, T_3, AC_1 (there are other possible orders). Both F_1 and AC_2 do not have any conflict and are therefore added first. F_1 is added before AC_2 because subprograms are preferred to components. Now, only entities with a conflict are left. Because subprograms are preferred to other kinds of entities, either F_2 or F_3 can be chosen. In this example, F_2 is selected first. When F_2 has been removed from the interference graph, F_3 does not have any conflict anymore and can be pushed onto the stack next. Now, either AC_1 or T_3 can be selected next because both have one conflict and there is no preference for components or types. The resulting stack is shown in Figure 4.

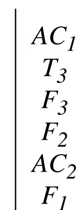

Figure 5. Resulting stack for coloring.

After all nodes have been removed from the interference graph, AC_1 is assigned to a new partition. Then T_3 is assigned to a new partition, too, due to its conflict with AC_1. F_3 is assigned to the partition of AC_1 because it is connected to AC_1 but not to T_3. F_2 cannot be added to the partition of AC_1 because it is in conflict with F_3. It can neither be added to the partition of T_3 because it has no connection to T_3. Hence, it is added to a new partition. AC_2 is added to the partition of F_3 because F_3 is connected to AC_2. Finally, F_1 can be added to the partition of AC_1 or T_3 because it has equal connections to both partitions. Hence, one possible partitioning of the candidate into subclusters without mutually exclusive elements is $\{AC_1, F_3, AC_2\}$, $\{T_3, F_1\}$, $\{F_2\}$.

5.5. Transforming Clusters into Candidates

The incremental versions of the techniques as proposed by the previous sections produce clusters that are to be transformed into atomic components. Principally, the clus-

ters fall into one of the following categories:

1. The cluster contains no component.
2. The cluster contains a single component. This is the case when an incremental technique wants the base entities of the cluster to be added to an existing component. Remember that the enclosing component is added to a cluster in lieu of a bound entity.
3. The cluster contains more than one component. This happens when at least one base entity can be added to more than one existing component.

For the first type of cluster, we can create a new candidate that contains all the elements of the cluster. In the second case, we add all elements to the component contained in the cluster. If clusters contain more than one component, it is not clear to which components the base entities of the cluster should belong. Therefore, such clusters are presented to the user as a whole and he or she can decide.

6. Set Operators

Since the results of the analyses are basically sets of components, they can be combined by set operations, namely, union and intersection.

The union is useful for techniques that produce very different kinds of components and is, therefore, especially suited to combine techniques that are restricted to one class of component. For example, *Delta IC* can only detect objects and *Part Type* only abstract data types. Applying the union operator to these two heuristics allows for detecting both kinds of components.

The intersection is used to reveal the agreement of two techniques: Only components detected by both techniques will be present in the resulting view. This gives us a higher confidence about the resulting components. A good example is the intersection of *Part Type* with *Internal Access*. *Part Type* assumes that the parameter of the part type in the signature is put into or retrieved from the parameter of the container type. This can only be the case when the parameter of the container type is internally accessed. The intersection is also useful when both techniques propose candidates that are too large. Large components of one technique can be refined by the other technique.

Applying the set operators is more than just uniting and intersecting the sets of components in the sense of set theory. For the intersection, for example, we can hardly expect that we find two exactly equal components by both techniques. A simple definition of intersection according to set theory, called **shallow intersection**, would, therefore, yield empty result views in most cases. Likewise for the simple union of component views, called **shallow union**, similar yet different components in the input views lead to many similar components in the output view. If this is presented to the maintainer, she has to check the overlapping parts of the similar components twice.

For example, in Figure 5, the shallow union of the sets of candidates *Result 1* and *Result 2* treats any candidate as an atomic set member and produces five candidates where four of them are very similar. The outcome of intersecting the two component views is even worse since the resulting view is empty. For set intersection, a component of one view is only in the resulting view when it has an exact match in the other view.

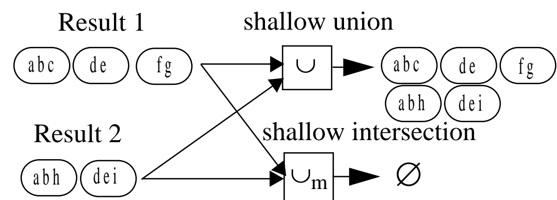

Figure 6. Shallow union and intersection example.

In order to avoid these effects, similar components should be treated as if they were equal. We consider two components A and B similar when

$$\frac{|\text{elements}(A) \cap \text{elements}(B)|}{|\text{elements}(A) \cup \text{elements}(B)|} \geq \Theta$$

where *elements* (A) denotes the elements of A and Θ is a user-determined threshold. If $\Theta = 1$, the two sets of elements must be equal.

Given similar components, we can unite or intersect their elements for the resulting component view of the union or intersection operator, respectively. In the case of two similar components for the union operator, this results in a single component that comprises all elements of the two similar components. In the case of the intersection, this leads to a single component that consists only of the elements that are in both components. Because not only the set of components as such are united and intersected but also the individual components themselves, these operations are called **deep union** and **deep intersection**, respectively.

6.1. Deep Union

The deep union operator unites the elements of two similar atomic components.

Note that overlapping atomic components can result from the union operator only. The intersection and composition do not yield overlapping atomic components other than those already produced by the applied techniques. Overlapping candidates are a problem when presented to a maintainer for validation because all overlapping candidates have to be investigated to decide where a given entity (in the overlapping part) belongs to. In the case of non-overlapping atomic components, the maintainer can simply accept or reject the entity at hand. However, this is only a

problem when the final result contains overlapping components. For intermediate results during combination, overlapping components are useful. This way, several alternative candidates can be investigated in parallel until a decision is made in the course of combination.

The result of the deep union for the example in Figure 5 for Θ=0.5 is shown in Figure 7.

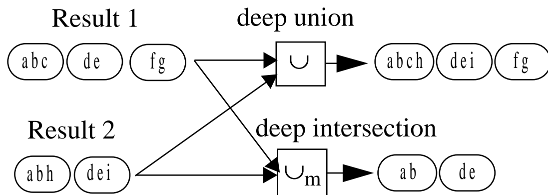

Figure 7. Deep union and intersection example.

6.2. Deep Intersection

The deep intersection operator intersects the elements of two similar atomic components.

The intersection does not make sense for every combination of techniques. If two techniques propose very distinct atomic components, the resulting set of atomic components may be empty. This is the case when techniques are combined that consider different kinds of entities. By a look at Table 2, which summarizes what kind of base entities are considered by the respective base technique, one can quickly decide which intersections do make sense. A ✓ in Table 2 means that a certain kind of base entity is regarded, ⊥ means that it is not regarded. A sensible intersection can be expected of those techniques that consider a common set of base entity kinds.

Table 2. Domains of the base techniques.

Technique	Subp.	Var.	Types
Same Module	✓	✓	✓
Part Type	✓	⊥	✓
Same Expression	✓	✓	⊥
Internal Access	✓	✓	✓
Delta IC	✓	✓	⊥
Similarity Clustering	✓	✓	✓
Type-Based Cohesion	✓	⊥	⊥
Data Bindings	✓	✓	⊥
Strongly Connected Components	✓	⊥	⊥
Dominance Analysis	✓	✓	✓

The result of the deep intersection for the example in Figure 5 for Θ = 0.5 is shown in Figure 7.

The intersection operator is adequate when two component views are to be combined. However, if the agreement among several techniques is to be established, the likelihood that the resulting view is empty increases by the number of techniques involved because all techniques have to agree to a component. When the results of several techniques are to be compared, a voting approach in which only a certain number of techniques have to agree but not necessarily all is a better solution. For details see [13].

7. Detection Strategy

The framework described in the previous sections has many degrees of freedom. Therefore, some guidelines should be given on how to use it successfully. The recommended strategy for component recovery consists of two main parts: Detection of components and then identification of the relationships among the components. Detection of components can be done in the following steps:

Step 1. Apply all connection-based analyses and *Strongly Connected Component Analysis* in parallel. Validate reasonable intersections of the results (see Table 2). Add negative information during validation when you find entities that should not be grouped together. **Rationale**: One gets only few promising candidates in the beginning which can form a point of crystallization later on. The added mutually exclusive information will break up larger candidates in the subsequent runs of the analyses.

Step 2. Apply connection-based approaches once again but one at a time. This time, they will leverage the information you have contributed and return clusters containing no entities that have already been grouped and no mutually exclusive entities. Validate the non-intersected results (thus, using less strict criteria). If particularly large candidates occur, refine them with other techniques by means of composition. If the candidates of one technique have been validated, run the next analysis. **Rationale**: The crystallization points of the first step are extended and new components are built that were dropped out by the intersection in the first step. Applying the techniques successively guarantees that all user information is respected by the analyses.

Step 3. The metric-based approaches are associated with parameters whose values may not be known in advance. The previous two steps lead to a set of components that can be used to automatically calibrate the metric-based approaches. Varying this calibration reveals further clusters. If a hierarchical metric-based clustering is used, one starts the validation at the leaves and then climbs up the tree toward the root until a metric value is reached that does not indicate sufficient confidence in the candidate anymore. **Rationale**: The connection-based techniques are based on fixed patterns and, therefore, will always yield the same candidates. The metric-based techniques allow more vari-

ability by changing their parameters.

Step 4. Finally, *Dominance Analysis* can be applied to find local utility functions of atomic components. **Rationale**: The dominance analysis for atomic components can only be applied when the atomic components exist in the first place. Hence, it can only be used late in the detection process. *Dominance Analysis* will detect the local entities that are not detectable by other approaches.

It is also usually helpful to restrict the search to one kind of atomic component at a time because the search criteria are mostly different. For this reason, it is possible to restrict all the analyses to a particular set of entity types. For example, if one searches abstract data types, global variables can be ignored.

Once the components have been detected, their relationships can be analyzed by applying *Strongly Connected Component Analysis* and *Dominance Analysis* to the collapsed graph. *Strongly Connected Component Analysis* yields sets of components that mutually depend on each other and *Dominance Analysis* reveals whether components are local to each other.

8. Related Research: Interactive Methods

In Section 3, research in automatic techniques for atomic component recovery has already been summarized. This section describes other approaches that integrate the user in the detection process and compares them to the semi-automatic method presented in this paper.

Müller et al. point out that architecture recovery cannot be fully automated [16]; thus, the role of the human software engineer constitutes a central and integral part of architecture recovery and, consequently, tools for architecture recovery should integrate the user. On the other hand, as much as possible should be automated. Therefore, the tool supporting their approach, namely, Rigi allows human intervention and offers automatic operations for subsystem detection, too. The available operations for subsystem detection are removal of omnipresent entities as well as composing by interconnection strength, common client/suppliers, centricity, and name. Furthermore, metrics can be used to asses the structure of the subsystem decomposition and exact interfaces of subsystems can automatically be derived. Moreover, because many analyses for architecture recovery are system-dependent, Rigi offers a scripting language that allows a maintainer to write his or her own clustering techniques.

Because the semi-automatic method proposed in this paper uses Rigi for visualization and user interaction, Müller's and my approach have a great deal in common. However, the semi-automatic methods focuses on atomic component recovery and, therefore, offers a wider selection of atomic component recovery techniques, while Müller's work is also aimed at hierarchical subsystems. Moreover, the only way of combining different techniques in the original Rigi is to apply the techniques successively, whereas the extended Rigi handles alternative results in parallel and offers deep intersection and deep union operators to combine these results.

Another extension of Rigi is Dali, developed by Kazman and Carrière [12]. Dali uses an SQL database to store information about systems to be reverse engineered. As a consequence, SQL can be used to specify clustering patterns. Kazman and Carrière distinguish two different levels of patterns: application-independent patterns and application patterns. Application-independent patterns are used to aggregate declarations according to the language semantics, like local variables with their enclosing functions or data and function members with their class (another type of low-level application-independent patterns filters noise introduced by insufficient tools for parsing and semantic analysis; these patterns are not needed when a capable C++ frontend is available). The purpose of application patterns is to group functions and classes to subsystems. These patterns leverage naming conventions or are enumerations of related elements.

The advantage of Dali is its support for SQL queries to specify clustering patterns; patterns can be written in a declarative manner and a language that is widely used. The patterns can be viewed as a specification of the system structure and used to re-generate the visualization of the system when the system has changed (in which case, the patterns have to be updated accordingly). Moreover, the patterns can be employed to group elements quickly instead of grouping them manually. In the case of patterns that are mere enumerations, however, this advantage is only limited. On the other hand, there is no analytic capability in Dali. Hence, the patterns have to be written by someone already familiar with the system and can neither be reused for other systems in many cases. Refinement of the results of a pattern is difficult because the user cannot interact with the system by direct manipulation since the refinement has to be done in the pattern. For the same reason, combinations of results are difficile. In principle, combining operators like deep union and intersection could be written in SQL as well using temporary tables and logical *or* and *and* operators, but the result would have to be expressed as an SQL query afterward to obtain the advantages of a separate specification of the system structure as SQL patterns.

The extended Rigi has also much in common with ManSART, a tool developed at MITRE that supports architecture recovery [26]. ManSART visualizes different views of the system that are directly derived from source code, such as task-spawning views, dataflow between procedures and

data files, and abstract data type views. In order to combine different views for presentation purposes, several operators are offered to the user. The purpose of the operators is to connect distinct base views at different levels of abstractions (e.g., a task-spawning view with the abstract data type view), whereas the extended Rigi offers operators to find agreements and differences of component views or to unite two views where all views are at the same level of abstraction. In order to establish correspondence among concepts in different views when views are combined, a containment relation is used by ManSART that is based on source positions of statements implementing the concepts. When the extended Rigi combines two views, it can consider the declarations contained in components because the corresponding components are at the same level of abstraction.

All approaches that have been presented in this section so far, including the one underlying extended Rigi, are primarily bottom-up approaches. The search starts at declarations extracted from the source code, which are then grouped together by automatic techniques and human judgement. Gall, Klösch, and Weidl complement bottom-up clustering by a top-down approach [6]. They also use bottom-up clustering heuristics that start at dataflow diagrams and then follow part-type relationships among data store entities of the dataflow diagrams and user-defined records and pursue data dependencies among record components. However, they go beyond a pure bottom-up process by using a domain model built by a human engineer (e.g., using the unified model language UML). The domain model describes the application concepts and their relationships. Part of the recovery process is to bind the domain concepts to the components found by the bottom-up phase. In order to establish this mapping, a similarity metric is used based on similarity of names of domain concepts and source code identifiers and on similarity of types [24]. Because it is not always possible to establish the mapping using the similarity metric only, the user is integrated in the binding process [6].

The domain model may be used to make the recovery process more goal-directed and may increase the chance to find components with application semantics. On the other hand, other programming concepts may be missed that are also necessary to understand the system or could be reused in another context. Moreover, the explicit domain model and a mapping from domain concepts to components in the source implementing these domain concepts is a valuable documentation. The bottom-up approaches discussed above also use a domain model — but it exists only in the head of the maintainer. However, building a domain model needs additional effort and the necessary degree of detail of a useful domain model is not known in advance.

9. Conclusions

This paper described a method in which human and computer interact to detect atomic components. The need for human intervention was indicated by a quantifying comparison of published techniques to detect atomic components [9]. This paper has described possible combinations of the analyses within this interactive framework. The preferred way of combining the base techniques is to offer simple operators to the user. This way, direct manipulation can be the way of interaction, the user has all the freedom to tailor the analyses, and new techniques can be quickly integrated. The only requirements for a new technique to be integrated are that the technique has to express its result as a component view, that is has to be incremental, and that its underlying clustering criterion should be expressed as a metric in order to use it for candidate ranking. The paper has also described how structure-based techniques can be extended to work incrementally.

Acknowledgments

I want to thank Hausi Müller and his group for making Rigi available. I also want to thank Erhard Plödereder, Thomas Eisenbarth, Hiltrud Betz, and Jean-Francois Girard for fruitful discussions.

References

[1] L.A. Belady and C.J. Evangelisti, "System partitioning and its measure", *Journal of Systems and Software*, vol.2, no 1., February, 1982.

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

[3] G. Canfora, A. Cimitile, A. De Lucia, G.A. Di Lucca, "A case study of applying an eclectic approach to identify objects in code," *Workshop on Program Comprehension*, 1999.

[4] Cimitile, A. and Visaggio, G., "Software salvaging and the call dominance tree", *Journal of Systems Software*, 28:117–127, 1995.

[5] N. Fenton and S. Pfleeger, *Software metrics: a rigorous and practical approach*, Pws Pub., 1997.

[6] H. Gall, R. Klösch, and J. Weidl, "Resolving uncertainties in object-oriented re-architecturing of procedural code", 7th int. Conf. on Information Processing and Management of Uncertainty in Knowledge-Based Systems, July, 1998.

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

[8] J.F Girard and R. Koschke, "A metric-based approach to detect abstract data types and abstract state encapsulation", *Conf. on Automated Software Engineering*, Lake Tahoe, 1997.

[9] J.F Girard and R. Koschke, "A comparison of abstract data type and object detection techniques", *Journal Science of Computer Programming*, Elsevier Science Publisher, to appear 1999.

[10] Hopcraft and Ullman, *Data structures and algorithms*, Addison-Wesley, 1983.

[11] D.H. Hutchens and V.R. Basili, "System structure analysis: clustering with data bindings", *IEEE Transactions on Software Engineering*, vol. SE-11, no. 8, August, 1985.

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

[13] R. Koschke, "Atomic architectural component detection for program understanding and system evolution", Ph.D. thesis. University of Stuttgart, to appear, 1999.

[14] C. Lindig, G. Snelting, "Assessing modular structure of legacy code based on mathematical concept analysis", *Proc. of the Int. Conf. on Software Engineering*, ACM Press, 1997.

[15] P.E. Livadas, T. Johnson, "A new approach to finding objects in programs", *J. of Software Maintenance: Research and Practice*, vol. 6, 1994.

[16] H. Müller, M.A. Orgun, and S. Tilley, and J.S. Uhl, "A reverse engineering approach to subsystem structure identification", *Software Maintenance: Research and Practice*, 5(4):181-204, December 1993.

[17] S. Patel, W. Chu, and R. Baxter, "A measure for composite module cohesion", 14th Int. Conference on Software Engineering, May, 1992.

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

[19] H. Sahraoui, W. Melo, H. Lounis, and F. Dumont, "Applying concept formation methods to object identification in procedural code", *Proc. of the Conference on Automated Software Engineering*, IEEE Computer Society Press, 1997.

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

[21] M. Siff and T. Reps, "Identifying modules via concept analysis", *Proc. Int. Conference on Software Maintenance*, IEEE Computer Society, 1997.

[22] G. Snelting, personal communication, Reengineering-Workshop, Bad Honnef, May, 1999.

[23] R.R. Valasareddi and D.L. Carver, "A graph-based object identification process for procedural programs," *Proc. of the Fifth Working Conference on Reverse Engineering*, October, 1998, IEEE Computer Society Press.

[24] J. Weidl and H. Gall, "Binding object models to source code: an approach to object-oriented re-architecturing", *Proc. of the 22nd Computer Software and Applications Conference*, IEEE Computer Society, August, 1998.

[25] A.S. Yeh, D.Harris, and H. Reubenstein, "Recovering abstract data types and object instances from a conventional procedural language", Proc. of. the *Second Working Conference on Reverse Engineering*, July 1995. IEEE Computer Society Press.

[26] A.S. Yeh, D.R. Harris, and M.P. Chase, "Manipulating recovered software architecture views", *Proc. of the Int. Conference on Software Engineering*, Association of Computing Machinery, 1997.