

Reverse Engineering Software-Models of Component-Based Systems

Landry Chouambe
Professional Services Organisation
Netviewer AG, Germany
landry.chouambe@netviewer.com

Benjamin Klatt and Klaus Krogmann*
Institute for Program Structures and Data Organisation
Universität Karlsruhe (TH), Germany
{klatt, krogmann}@ipd.uka.de

Abstract

An increasing number of software systems is developed using component technologies such as COM, CORBA, or EJB. Still, there is a lack of support to reverse engineer such systems. Existing approaches claim reverse engineering of components, but do not support composite components. Also, external dependencies such as required interfaces are not made explicit. Furthermore, relaxed component definitions are used, and obtained components are thus indistinguishable from modules or classes.

We present an iterative reverse engineering approach that follows the widely used definition of components by Szyperski. It enables third-party reuse of components by explicitly stating their interfaces and supports composition of components. Additionally, components that are reverse engineered with the approach allow reasoning on properties of software architectures at the model level. For the approach, source code metrics are combined to recognize components. We discuss the selection of source code metrics and their interdependencies, which were explicitly taken into account. An implementation of the approach was successfully validated within four case studies. Additionally, a fifth case study shows the scalability of the approach for an industrial-size system.

Reverse engineering, Components, Modeling, Software performance

1 Introduction

Reverse engineering as defined by Chikofsky and Cross [4] can be applied in various fields, such as comprehension of legacy software systems or for planning the extensions of existing systems. In the latter case, if existing systems are extended by new functionality, it is desirable to reason on the quality properties of the new software architecture. Extra-functional properties like performance are of interest if additional concurrent users are expected for the extended system or if a cache is introduced.

In recent years, a lot of software is developed using component technologies like COM, EJB, or CORBA.

Component-based software has proven to be especially suited for reasoning on the quality of an architecture at model-level [2, 9, 19]. To the best of our knowledge, existing reverse engineering approaches for component-based software architectures (e.g. [12, 17]) do not support creating models of components that enable reasoning on quality properties: Either interfaces are not made explicit or composite components (that are required for large software systems) are not supported. Such relaxed component definitions make components indistinguishable from OO concepts like modules or classes.

Thus, it would be desirable to have a reverse engineering approach that follows a stricter, widely used definition of components introduced by Szyperski [25, p. 548]: “A component is a unit of composition with contractually specified interfaces and explicit context dependencies only. Context dependencies are specified by stating the required interfaces [...]. A component can be deployed independently and is subject to composition by third parties.” This definition implies that components have interfaces to specify provided and required services. Their communication with other components is limited to these interfaces only. Components can be composed from other components. Components conforming to this definition enable analysis of software architectures. We will discuss a more precise component definition, further assumptions on components, and its effects on reverse engineering in the next section.

The approach presented in this paper is able to reverse engineer component-based software architectures, including Java Enterprise Edition based systems that conform to the component definition from above. It takes the source code of an existing software system, creates an instance of a meta-model describing components, composite components (components built from components), interfaces, and the relation between interfaces and components (i.e. whether interfaces are provided or required by a component). As the target model is formally defined within an meta-model, this enables reasoning on the impact of extensions to the performance of the architecture [2].

Our approach is based on source code metrics specifically selected for component-based software systems. Metrics are a) weighted and b) combined, while c) respecting interdependencies among them based on empirical evaluation. Such an interdependency is e.g. to consider similar class names only if classes are coupled. Our prototype implementation utilises the state-of-the-art reverse engineering tool “Sotograph” [24] for a static analysis of source code and then outputs an instance of the Palladio Component Model (PCM) [22], which enables predictions of performance properties of a software architecture such as throughput and response time.

The contribution of this paper is an iterative reverse engineering approach for component-based software architectures, following a stricter definition of components by Szyperski [25]. The approach is suited to reverse engineer multiple abstraction levels of components (composite components). An empirical evaluation led to a selection of suitable source code metrics and weightings for them while respecting interdependencies among metrics to reflect component properties. An iterative and interactive process embeds our approach to be applicable for large applications. We successfully evaluated a prototype implementation of the approach with five case studies ranging from small software systems (8,000 LOC) to industrial-size systems with more than 600,000 LOC.

The remainder of this paper is structured as follows: In Section 2, we present the domain for which our reverse engineering approach is suited, Section 3 discusses related work, Section 4 details the concepts of our reverse engineering approach while Section 5 gives insights into the prototype implementation “ArchiRec”. Then, in Section 6 we discuss the settings and results of our evaluations within five case studies, show the limitations and assumptions of our approach in Section 7, and finally conclude the paper in Section 8.

2 Context and Requirements

To understand the reverse engineering approach presented in this paper, we first introduce its context. The approach is aligned with Palladio [2], an approach that allows the prediction of performance attributes for component-based software architectures expressed in the *Palladio Component Model* (PCM) [22]. Model instances that are reverse engineered with our approach are “specification models” [3, p. 22] in terms of Cheesman and Daniels. A model instance of the PCM consists of a) a structural description of the static component architecture (components, interfaces, and their wiring) and b) a behavioural description, comparable to UML activity charts. Prediction methods from Palladio are then applied to those model instances. Therefore, model driven techniques [26] need to be applied to model instances. In this paper, we focus on the static part of the

PCM, which is the target model of the reverse engineering approach presented in this paper.

People using the PCM approach are for example reasoning on extensions of existing architectures or evaluate “what-if-scenarios” by changing an existing architecture at the model level. This implies a strict and limiting component definition:

- components are solely communicating via their interfaces (separated between provided interface and explicit required interfaces)
- components can be part of composite structures (composite component)
- interfaces define a number of services, serving as a contract between components
- services use data type arguments only, not other kinds of objects known from object-oriented languages, or passed references
- components can be reused independently by third parties by explicitly stating their environmental dependencies

The component definition of the PCM is in line with [25]. The abstract syntax of modeling constructs is defined within an ECORE-based [6] meta model.

As the reverse engineering approach aims at business information systems, it is appropriate to support Java EE components Enterprise Java Beans (EJB), which indeed do not satisfy the strict component definition (e.g. EJBs do not allow composite components), but still can serve as input. Therefore, the intended approach should support specifics like EJB deployment descriptors.

Still, reverse engineered component-based architectures should follow common principles like cohesion and coupling [18] and abstractness [15]. The intended iterative approach should allow human interaction while being configurable to reflect specifics of a reverse engineered system.

3 Related Work

Several reverse engineering approaches exist for component-based software systems. While existing approaches [7, 12, 17] often assume that components are modules or classes, the approach presented in this paper has a more limited domain than other reverse engineering approaches: It focuses on the narrower and stricter component definition (given in the previous section) that allows component-based software engineering for predicting its properties and includes high-level composite structures.

Koschke [13, pp. 351] gives an overview on reverse engineering approaches for software-architectures. In his PhD thesis, Koschke deals with the detection of components, but has a different definition of them: His approach is part of the Bauhaus Suite [1], which offers component detection,

but has a more relaxed definition of components. Basically, “logical connected elements” from the source code are considered as components. Out-of-the-box Bauhaus does not support Java EE components (e.g. deployment descriptors or Enterprise Java Beans structures).

The voting approach Koschke presents in his PhD thesis [13, pp. 301], is comparable to the way in our approach metrics are combined, except an important difference: Koschke sums up the “agreement” of *all* heuristics. If then a certain threshold is passed, recognized components are added to existing “atomic component” clusters. In our approach thresholds are not applied to sums of all metric values, but only to selected metrics, expressing interdependencies of metrics (see example in introduction). Still, the approach relies on the relaxed component definition by Bauhaus.

Favre et al. present an approach claiming to be component based [7]. Their reverse engineering target is a meta model called “Object Model”, concurrently formalising their concept of a component. Like in our approach, the meta model is formalised, though Favre et al. use UML instead of ECORE in our case. Their components are built by a decorator pattern [8] but do not allow composition of components, which we suppose to be an elementary property of components. Favre et al. utilise OCL to check constraints as the PCM does. They present a java-based tool chain that includes visualisation options, but give little details on the reverse engineering approach itself.

Keller et al. [12] deal with the detection of so-called “design components” and evaluated their C++-based approach within a case study of three systems. Their component definition includes “patterns, idioms, application-specific solution[s]” [12, p. 227] and “package[s] of structural model descriptions together with informal documentation” [12, p. 227] and is therefore much broader than the definition in our approach. Keller et al. use pattern-recognition for identifying components. Supported patterns (template method, bridge, or factory method; cf. [8,23]) are not related to components according to the definition from Section 2.

Muller et al. [17] aim at recovering architectural design information. They support composite subsystems following the principle of cohesion and coupling (cf. [18]). The approach is implemented in the Rigi tool. Components are more broadly defined as aggregations of variables, procedures, modules, and subsystems, allowing low-level components. As within our approach, Muller et al. make required interfaces of their components explicit within so-called “exact interfaces”: only those methods/interfaces of a subsystem that are really required are listed. For example, an import of a complete package is not regarded as a requires relation for the whole package. This is in line with the required interfaces in our approach.

Mitchell and Mancoridis [16] use a clustering approach (utilising hill climbing and simulated annealing) to modularize according to cohesion and coupling. To optimize performance they use heuristics for modularization. However, they do not cover component characteristics. While our approach distinguishes between inheritance, call reference, number of calls, and kind of call relations, their approach is based upon directed graphs that does not keep such differences. Like our approach, theirs is iterative and supports quantitative distance metrics.

Our approach, presented in this paper, bases on the work of Chouambe [5].

4 Our Approach to Component Identification

For this paper, we assume that an experienced software architect is able to reconstruct component-based software architectures better than an automatic approach. Nevertheless, a software architect would perform better with tools supporting the reverse engineering process. The tool presented in this paper suggests possible components (basic and composite structures), the software architect then interacts with the tool to refine them.

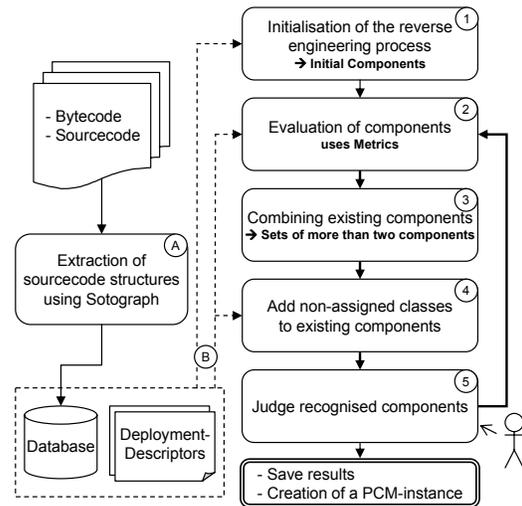


Figure 1. Iterative Reverse Engineering Process

The ArchiRec-approach uses information about the relationship between classes and interfaces as well as a set of metrics to reconstruct the component architecture of a software system. Starting from the relationships between classes and interfaces, we obtain initial components, make different hypotheses for their compositions in new components, and then gain new abstractions. The latter is the goal for each iteration in this reconstruction approach. Each it-

eration gains a new abstraction based on the result of the previous iteration.

4.1 Overview on the major Reverse Engineering Steps

The starting point is a list of classes and interfaces and the relationships between each other in the analysed software system (extracted from given source- and bytecode in step A, Fig. 1). Out of this information, the first components are deduced (step 1). In the following, we refer to them as *initial components*. Initial components make up the first (and lowest) abstraction of the analysed software system. Usually there is a large number of initial components depending on the number of classes implementing interfaces or calling their methods. The abstraction level of such components is low and their interdependences can be high.

After finding initial components, this approach uses metrics to evaluate them regarding their composition in new composite components (step 2). Each metric covers one or more aspects of a component. The value of a metric is called *score*. To obtain new components, this approach states different *hypotheses*, i.e., a potential composition of existing components. For each of these hypotheses all metrics are computed. Weighted scores are then combined to an overall score for this hypothesis. The highest scored hypotheses are turned into new components (step 3), representing a new abstraction. Non-assigned classes are added to these components (if reasonable), depending on their coupling with classes and interfaces from existing components (step 4).

The fifth and last step includes judging the new gained abstraction and the overall result of the reconstruction by the user. The user can stop the reconstruction or launch a new iteration. Only the first iteration performs the initialization. Every following iteration will start with step two and the stating of new composition hypothesis.

A detailed discussion of these steps is done in the following sections.

4.2 Finding Initial Components, Step 1

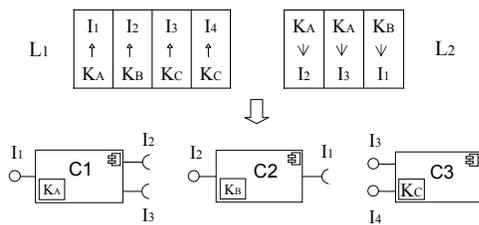


Figure 2. Initial components

The reverse engineering starts with an initialization task in which the initial components are recognized. We first create two lists L1 and L2 of tuples consisting of a class

and an interface. Each tuple of L1 consists of a class and an interface implemented by this class, and each tuple of L2 consists of a class and an interface used by this class. An example of these two lists is shown in Fig. 2. K stands for class, I stands for interface and the arrows represent the “implements” respectively the “references” relationship according to UML class diagrams.

Each item of these lists is considered as partial component, made from one class and one interface. The ones from L1 having only provided interfaces, while items from L2 have only required interfaces. All of these components are put together in a new list. Thereby the partial components having the same class – independent from the list the component is coming from – are joined. Each of these initial components is made from one class (the so-called dominant class (cf. [8, pp. 185]) and one or more provided or required interfaces. For example, the four items from L1 and three from L2 are joined into three components as shown in Fig. 2.

We then finally iterate over the list of the obtained components and assign those classes to them that are implemented in the same files as their dominant class. Nested classes in Java are an typical example for this. The resulting components are initial components. They represent the first and lowest abstraction gained in this approach.

4.3 Composition hypotheses: Discussion of Complexity

Starting from the initial components, we gain new and higher abstractions of the analysed software system. Each new abstraction-level is a set of composite components, obtained by composing the components from the previous abstraction-levels. For example starting with three initial components, C_1 , C_2 , and C_3 results in the following composition hypotheses: $\{C_1, C_2\}$, $\{C_1, C_3\}$, $\{C_2, C_3\}$, and $\{C_1, C_2, C_3\}$.

Thus, for 3 components we have 4 ($= 3 + 1$) combinations, for 4 components we have 11 ($= 6 + 4 + 1$) combinations and for 8 components we have 247 ($= 28 + 56 + 70 + 56 + 28 + 8 + 1$) combinations. Generally, it is a combination of k over n . The number of hypotheses we then have to consider (and score with metrics) according to this method is very high. It leads to a computational problem not resolvable in reasonable time.

Hence, our approach will not consider all composition hypotheses of components, but just pairs. That means, for $n \geq 2$ components only $\binom{n}{2} \Rightarrow O(n^2)$ component hypotheses are going to be considered. For 3 components we have 3 hypotheses, for 4 components 6 hypotheses and for 8 components only 28 hypotheses. Though the approach considers component hypotheses made from pairs of com-

ponents only, the approach recovers components from more than two components as we will point out in Section 4.6.

In the next two sections we address the evaluation of composition hypotheses (pairs of components), especially the metrics used to do this (Section 4.4), and the process (Section 4.5) for weighting the scores of these metrics.

4.4 Selected Metrics

Our approach uses metrics to rate the composition hypotheses. The list of the metrics we use for this purpose is shown in Table 1. We selected metrics for the approach to reflect component properties (according to our definition from Section 2) as good as possible. However, the approach is not limited to the metrics from the list but can be extended by additional metrics. After introducing the metrics we will discuss their combination separately.

Table 1. Metrics and covered aspects of components

Metric	aspect of component
Abstractness	constitution
Instability	constitution
DMS	constitution
Name Resemblance	naming of classes and interfaces
Interface Violation	communication through interfaces
Coupling	dependences / interconnection
Package Mapping	organisation in packages
SLAQ	slice-layer architecture style
Subsystem component	· organisation in packages · slice-layer architecture style

The first three metrics (Abstractness, Instability and Distance from the Main Sequence, DMS) are object oriented metrics from R. Martin [15], which we adapted to components. The other originate from our previous work [5].

The *Abstractness* [15, p. 6] of a component is the ratio of the number of its abstract classes and interfaces to the total number of classes and interfaces of this component. It can be considered as an indicator for the encapsulation of a component. The extend to which classes depend on component-internal and -external classes and interfaces is reflected by the *Instability* [15, p. 6]. It is calculated based on the dependencies between classes and interfaces existing inside and outside of a component. The *Distance from the Main Sequence* [15, p. 7] uses the previous metrics. It reflects the quality of the combination of Abstractness and Instability of a given component: Components that are neither abstract nor stable (indicating little communication abilities) or fully abstract and instable (little implementation and little encapsulated) are unwanted.

$$\text{Abstractness: } A = \frac{N_a}{N_c} \quad (1)$$

N_a : number of interfaces and abstract classes of the component

N_c : number of interfaces and classes of the component

$$\text{Instability: } I = \frac{C_e}{C_a + C_e} \quad (2)$$

C_e : number of outgoing dependences

C_a : number of incoming dependences

$$\text{Distance from the Main Sequence: } DMS = |A + I - 1| \quad (3)$$

A : Abstractness

I : Instability

The *Name Resemblance* [5, pp. 54] is the ratio of the number of classes and interfaces of a component which have similar names and the total number of classes and interfaces. Evaluations during our cases studies showed that similar names of classes and interfaces often indicate that they belong to the same component.

$$\text{Name Resemblance: } SN = \frac{N_{sn}}{N_c} \quad (4)$$

N_{sn} : number of classes and provided interfaces of a component having similar names

N_c : total number of classes and provided interfaces of this component

The *Interface Violation* [5, pp. 55] calculates the extent to which components are following pure interface communication. It is defined for two components C_X and C_Y , as the ratio of the number of references from classes of C_X to classes of C_Y (rather than interfaces), to the number of references from classes of C_X to classes and provided interfaces of C_Y . This metric allows punishing every communication between components, taking place outside of their interfaces.

$$\text{Interface Violation: } IV_{XY} = \frac{R_c}{R_{XY}} \quad (5)$$

R_c : number of references from classes of C_X to classes of C_Y

R_{XY} : number of all references from classes of C_X to provided interfaces of C_Y

As components are assumed to have low coupling, communicating solely via their interfaces, the approach uses *Coupling* [5, pp. 58]. It is defined for two components C_X and C_Y , as the ratio of the number of references from classes of C_X to classes and provided interfaces of C_Y , to the number of all outgoing references from classes of C_X .

$$\text{Coupling: } C = \frac{R_Y}{R} \quad (6)$$

R_Y : number of references from classes of C_X to classes and provided interfaces of C_Y

R : number of all references, going out from classes of C_X

The *Package Mapping* [5, p. 59] addresses the mapping of a component to packages. It judges whether classes and provided interfaces of a component are implemented in the same package (or sub package) or not. In our cases studies we found out that classes and interface of a component are often placed in the same package.

$$\text{Package Mapping: } P = \begin{cases} 1, & \text{if all classes and provided interfaces} \\ & \text{belong to the same package} \\ 0, & \text{otherwise} \end{cases} \quad (7)$$

The *Slice Layer Architecture Quality* [5, pp. 60] checks the similarity of the architecture of the analysed software system with an architecture organised in slices and layers. The architecture of a software system is organised in slices and layers if it is for example partitioned into business and technical aspects. This is a particular architecture style, encouraged in [10] [27, p. 29]. It defines a clear mapping of an architecture on the package level. The Slice Layer Architecture Quality simply checks, whether the packages organisation of the analysed software system adheres to this concept. Components can then be identified within those slices and layers.

Slices and layers that occur at the highest abstraction level of an architecture are the so-called “natural subsystems”, lying well-defined in one slice and one layer. The last metric, *Subsystem Component* [5, pp. 61], assumes that a system is organized in slices and layers and checks the mapping of a given component into one of its natural subsystems.

$$\text{Slice Layer Architecture Quality: } SLAQ = \frac{S_l}{S_e} \quad (8)$$

S_l : number of existing natural subsystems

S_e : number of expected natural subsystems

$$\text{Subsystem Component: } SLA_{sub} = \begin{cases} SLAQ, & \text{all classes of a component} \\ & \text{are implemented} \\ & \text{in a unique natural} \\ & \text{subsystem} \\ 0, & \text{otherwise} \end{cases} \quad (9)$$

4.5 Combining Metrics, Step 2

For each composition hypothesis, we then have a set of six scores (x_0, \dots, x_5), the values of the Distance from the Main Sequence, Name Resemblance, Interface Violation, Coupling, Package Mapping, and Subsystem Component (see Fig. 3). The Abstractness and Instability are indirectly used through the Distance from the Main Sequence and Slice Layer Architecture Quality.

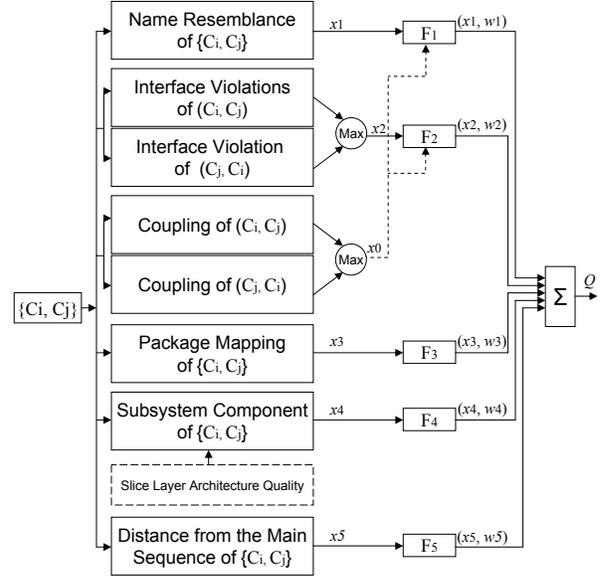


Figure 3. Combining Metrics

Before combining the scores, each one is weighted as we will point out below. Not every metric is directly considered within the overall score, as we take care of some interdependencies:

a) The Name Resemblance and the Interface Violation depend on the score of the Coupling (see Fig. 3). If, for example classes of two components have identical names (Name Resemblance) but no relation at the code level (Coupling), they would not provide a good composite component. Hence, only if a certain threshold of coupling is exceeded, Name Resemblance is used for the overall score.

b) If components from a composition hypothesis are not coupled at all, it is of little interest whether there are interface violations. Thus, the Interface Violation influences a composition of components only in a positive way if these components are first of all highly coupled. For low coupling the value of Interface Violation is ignored and only other metrics decide on composition. The Interface Violation is considered if the Coupling is high enough (typically 50%; with 0% minimum coupling and 100% maximum coupling).

c) The closer the architecture of the analysed software system comes to the slice-layer architecture style, the more we consider two components from a composition hypothesis as a composite component because of the Subsystem Component. As the Subsystem Component bases on the Slice Layer Architecture Quality, we do not consider the latter in the overall score.

To compute the *overall score* of a composition hypothesis, the approach uses the following formula:

$$\text{Overall score: } Q = \frac{\sum_{i=1}^4 x_i w_i - x_5 w_5}{\sum_{i=1}^5 w_i} \quad (10)$$

$$0 \leq x_i \leq 1 \text{ and } 0 \leq w_i \leq 100.$$

Where the x_i (with $1 \leq i \leq 4$) are the scores of the Name Resemblance, Interface Violation, Subsystem Component and Package Mapping with their corresponding weights w_i . x_5 and w_5 are the score and the weight of the Distance from the Main Sequence. In contrast to the other metrics, a higher score indicates a higher probability that the analysed components do not belong together. So, its score should be closer to 0 for good composition hypothesis. The score of the Coupling is only used for weighting of metrics. We obtained and improved the weights experimentally, using real software systems. (Detailed information about this can be found in [5, pp. 66].)

4.6 Getting new Abstractions, Step 3

The overall score of each composition hypothesis ranges from -1 to 1 . The higher the score, the higher is the probability that the two components of the corresponding composition hypothesis constitute a good composite component. After scoring of hypotheses, we can simply turn each hypothesis having a good score into a concrete component. But, we consider more than one composition hypothesis at once and try to turn them into a single new component. In other words, we score pairs of components (due to the high computation complexity of metrics), but take more than two components to build up each composite component.

In order to do this, we first define a threshold for the scores, typically 0.5 . Every hypothesis with a score smaller than this threshold is taken as a bad hypothesis. For each component C_1 , we create a list S of all components building good hypotheses with C_1 . We go through the list of the good hypotheses without C_1 , in decreasing order by score, and check the existence of a hypothesis built by two components of the list S . If we can find such hypothesis, we then insert its components in a third list T . With C_1 and the components of T , we then create a new composite component. All hypotheses with components of T are then removed from the list of good hypothesis.

If we have for example the following different composition hypotheses $\{C_1, C_2\}$, $\{C_1, C_4\}$, $\{C_1, C_5\}$, $\{C_2, C_4\}$ and $\{C_3, C_5\}$, this can result in the composite components $C'_1 = \{C_1, C_2, C_4\}$ and $C'_2 = \{C_3, C_5\}$.

The set of new components constitute a new and higher abstraction of the analysed software system. Each composite component adopts all provided and required interfaces of its internal components. Interfaces provided and only required by internal components are going to taken as internal

interfaces. It keeps references to the internal components, which allow tracing back the composition of the components down to the source code level.

4.7 Completion of Components, Step 4

Up to now, our approach only considers classes implementing or using interfaces. After the composition (Step 3), the approach handles the remaining classes. Each of these “free” classes is assigned to a component it is related to, if there is no other component having a relation with it. In this step the approach also assigns a class to a component, if this component contains a class implemented in the same file as this class (e.g. nested classes in Java).

4.8 Appreciation, Step 5

After the automatic completion of found components, ArchiRec gives the user the opportunity to change the assignments of classes to components, to meet his comprehension of the system. The rule we use for the completion is very simple and does not carry out many probable cases. In uncertain situations, we prefer to let the user do this assignment. One of the important actions the user can do now, is naming the components. This can facilitate the comprehension of the system after the reconstruction, otherwise components are named automatically with a unique number.

The user can also launch a new iteration, which will start out from the creation of new hypothesis based on the actual set of components. The reconstruction ends, if no new components could be found during an iteration or if the user considers the reverse engineered components to be good enough.

5 ArchiRec: Prototype Implementation

We provide a prototype implementation of our reverse engineering approach. This prototype is a stand-alone application, called ArchiRec, implemented in Java. It has a Swing based user interface and uses the Prefuse library [20] for the graphical representation of classes, interfaces, and components.

During the reconstruction, an interaction with the user is required after each iteration, as he has to validate the new abstraction and launches a new iteration (if necessary).

The input of ArchiRec is a database with the extracted source code of the system that should be reverse engineered. The database is provided by the tool Sotograph [24]. Sotograph parses the bytecode and source code of a software system and extracts information. Among others, all dependencies between source code artifacts, as we need for our approach at the level of directories, packages, files, classes and methods are stored in a relational database and later

used by ArchiRec (Fig. 1 Step A). Support of EJB deployment descriptors enables reverse engineering of EJB-based systems.

At the end of the reconstruction, ArchiRec provides a component-based architecture, made up of basic components and composite components. On demand, ArchiRec can generate an instance of the Palladio Component Model.

6 A Case Study using ArchiRec

We validated the prototype implementation “ArchiRec” of our approach with multiple software systems each having different characteristics.

Mitchell and Mancoridis [16] aim at software clustering and argue that is not required to have a “reference” decomposition, while Koschke [14] states that a reference decomposition is required to point out the strengths and weaknesses of individual clustering approaches. For the smaller projects we manually reverse engineered components from the code and then compared the number of components with automatically reverse engineered components. For the larger ones we used an inverse approach and analyzed whether the detected components are reasonable or not. Only for one project (“CoCoME”) an additionally documented architecture has been available for comparison.

The execution time has been measured on a notebook with a 2.33 GHz Core2 Duo CPU and 4 GB RAM. Tests on another laptop with 2.2 GHz CPU and 2 GB RAM came up with nearly the same results. We measured the execution time of the first iteration because it can be performed without any manual intervention. Furthermore the execution times of subsequent iterations for the different projects are not as well comparable as for the first iteration. Their complexity differs a lot depending on the results of the first iteration.

Rubis (rubis.objectweb.org) is an open source auctioning platform for benchmarking application design patterns and application servers. While there are implementations for different platforms available, the case study has been created with the one based on Enterprise Java Beans.

Table 2. Rubis case study data

LOC:	8,202
Classes:	41
Interfaces:	34
Enterprise JavaBeans:	17
Prefixes:	SB_
Suffix:	Bean, Home
Detected Components:	17
Performed Iterations:	2
Execution time (1st iteration):	1.5 sec

A manual analysis of the code detected a repeated pattern of a session bean that uses a home and a remote interface as well as a servlet associated with it. The identification of

the pattern is supported by the naming of the classes and interfaces. 16 of those components could be found and all classes not included are commonly used utility classes. The manual reconstruction did not reveal any higher-level composite component structures.

17 components have been detected using ArchiRec. 16 of them are identical with the manual detected ones. One component misses a class with a similar name as the included elements but is not referenced by any class of the detected component. The class itself only references one utility class. The result represents a reasonable component model after the first iteration. Additional iterations come up with only one composite component for the complete system. The fact that the number of Enterprise JavaBeans and detected components are the same is a coincidence.

Ohioedge (www.ohioedge.com) is a web-based open source customer relationship and business process management software for medium-to-large size organisations.

Table 3. Ohioedge case study data

LOC:	78,516
Classes:	249
Interfaces:	132
Enterprise JavaBeans:	63
Prefixes:	none
Suffix:	EJB, Bean, Home, PK, Delegate
Detected Components:	13
Performed Iterations:	5
Execution time (1st iteration):	31 sec

Because of the size of the software a manual code analysis could be performed only based on the code artifact names and a few code insights. This did not come back with a clear set of components. The reconstruction started with 119 components and was stopped after 5 iterations with 13 detected components. The composed components have been valued as a reasonable combination by a manual analysis. A comparison of the reverse engineered model against an existing one was not possible because no model is available in the documentation.

CoCoME (www.cocome.org) is an example implementation of a distributed trading system developed as an academic project of the universities of Clausthal, Karlsruhe, Milano, Prague, and Munich to compare and evaluate different component models. The current implementation is based on Java but does not make use of Enterprise JavaBeans.

A manual code analysis did not take place because of the satisfying in-depth documentation of the system. With every iteration the reconstructed architecture approximates the documentation but remains on a much more detailed level. All detected components are documented as basic components (full recovery). As an example, the components “reporting” and “store” are documented as the composite com-

Table 4. CoCoME case study data

LOC:	9,521
Classes:	126
Interfaces:	21
Enterprise JavaBeans:	0
Prefixes:	none
Suffix:	If, Event, Impl, Factor, GUI, TO
Detected Components:	18
Performed Iterations:	3
Execution time (1st iteration):	8 sec

ponent “GUI” but have no coupling in the source code except of a high level package name, as the GUI is a pure logical component. The reconstruction takes this into account and does not come up with the composite component.

openArchitectureWare (oAW) is an open source platform for model-driven software development. It supports the transformation between models and the generation of code out of them. The manual reconstruction of the software did not identify a component-based architecture, hence oAW was used to check the treatment of improper input architectures. ArchiRec did not detect a reasonable component-based architecture. This has to be interpreted as a valid result for our reconstruction approach.

Industrial Project. Itemis, an application engineering and consulting company based in Germany provided the sources and models of a customer project for the tool evaluation. The application had a total of 646,527 LOC, 2613 classes, and 452 interfaces allowing to check the scalability of our approach. Analysing the large system required a significant higher amount of computing time. The first iteration took about 7 hours but the processing was stable. Up to now, we did not evaluate the analysis results due to time constraints. However, we gained a lot of information about the current bottlenecks of the tool. ArchiRec is currently being refactored to be able to process software of this scale faster.

The evaluation within the case studies successfully showed that it is possible to reverse engineer composite component structures, following a non-OO definition of components.

7 Limitations and Assumptions

It is a basic assumption for our approach that the analysed software was designed with components in mind or that the implementation contains some of them. For software that is not component-based (as it is the case with openArchitectureWare), ArchiRec does not provide any means to recommend stopping further reverse engineering iterations. If a user is not able to recognize this situation by himself, ArchiRec may perform more iterations than needed.

The used metrics have a complexity that limits the scalability of the reverse engineering approach. The calculation of the metrics has been optimized during the case studies. As they represent the bottleneck of the process, accelerating their computation will be addressed in future work.

At the moment, the tool only interprets Enterprise JavaBean deployment descriptors as meta data to support reverse engineering. There are different kinds of meta descriptors in the state-of-the-art frameworks such as Spring that should be supported.

There is just a basic support for renaming components in the current implementation. The user can request ArchiRec to enumerate the components. This has to be improved to build better understandable component models.

8 Conclusions

We presented an iterative reverse engineering approach for component-based software architectures following a stricter definition of components (compared to several previous approaches) as proposed by Szyperski [25]. Those components include for example composite structures to support multiple levels of hierarchy. Thereby, we enabled the use of the reverse engineered models as base for prediction of quality attributes and reasoning on software architectures in the following scenarios:

- comprehension of legacy software systems
- documentation of component-based software systems
- planning the extension of existing systems
- support for playing “what-if-games” wrt. quality attributes of the software architecture
- identification of reusable components

Our approach bases on source code metrics. The selection of metrics reflecting component properties was discussed in detail. We introduced interdependent metrics expressing interrelations between basic metrics. Adjustment of metrics weighting was evaluated empirically within five case studies with software systems from 8 KLOC up to 646 KLOC industrial-size systems. We found the approach to be applicable within an iterative and interactive reverse engineering process, still being largely automated. Comparing the quality of manually and automated reverse engineered models of the systems under investigation showed a satisfactory result.

During the case studies we learned that we should still improve the recognition of logical components, which are not present in the code itself, but often are available through package names. For example, CoCoME had several of those components.

For future work we plan to integrate our approach with the work of Kappler [11]. So far, our approach does not include details on the internal of components, namely behavioral specifications that are required to enable automated

prediction of quality attributes. ArchiRec is planned to reverse engineer the static component structure, while the approach of Kappler will be used to reverse engineer behavioral specification. Both approaches together will enable the prediction of performance properties with the Palladio approach [22].

As we pointed out in the Limitations section, further improvements concerning ArchiRec itself will primarily strike performance issues. Currently there is a strong connection to Sotograph. ArchiRec should provide an interface to be open for other code extraction products. We will replace the Sotograph data layer by an open source tool data layer using Recoder [21].

Acknowledgement

The authors would like to thank “itemis GmbH” for supporting this work and providing the industrial-size case study and “Software-Tomography GmbH” for making available Sotograph.

References

- [1] Bauhaus homepage. <http://www.iste.uni-stuttgart.de/ps/bauhaus/>. last retrieved 2007-10-30.
- [2] S. Becker, H. Koziolok, and R. Reussner. Model-based Performance Prediction with the Palladio Component Model. In *Proceedings of the 6th International Workshop on Software and Performance (WOSP2007)*. ACM Sigsoft, February 5–8 2007.
- [3] J. Cheeseman and J. Daniels. *UML Components: A Simple Process for Specifying Component-based Software*. Addison-Wesley, Reading, MA, USA, 2000.
- [4] E. J. Chikofsky and J. H. Cross. Reverse engineering and design recovery: a taxonomy. *IEEE Software*, 7:13–17, January 1990.
- [5] L. Chouambe. Rekonstruktion von Software-Architekturen. Master’s thesis, Faculty of Informatics, Universität Karlsruhe (TH), Germany, May 2007.
- [6] Eclipse Foundation. Eclipse modeling framework homepage. <http://www.eclipse.org/modeling/emf/>. last retrieved 2007-10-24.
- [7] J.-M. Favre, F. Duclos, J. Estublier, R. Sanlaville, and J.-J. Auffret. Reverse engineering a large component-based software product. In *Fifth European Conference on Software Maintenance and Reengineering*, pages 95–104, Lisbon, Portugal, March 2001. IEEE.
- [8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, Reading, MA, USA, 1995.
- [9] V. Grassi, R. Mirandola, and A. Sabetta. From Design to Analysis Models: a Kernel Language for Performance and Reliability Analysis of Component-based Systems. In *WOSP ’05: Proceedings of the 5th international workshop on Software and performance*, pages 25–36, New York, NY, USA, 2005. ACM Press.
- [10] hello2morrow GmbH. SonarJ White Paper. <http://www.hello2morrow.de/download/SonarJWhitePaper.pdf>. last retrieved 2007-10-13.
- [11] T. Kappler. Code-Analysis Using Eclipse to Support Performance Prediction for Java Components. Master’s thesis, Faculty of Informatics, Universität Karlsruhe (TH), Germany, Sept. 2007.
- [12] R. K. Keller, R. Schauer, S. Robitaille, and P. Pagé. Pattern-based reverse-engineering of design components. In *ICSE ’99: Proceedings of the 21st international conference on Software engineering*, pages 226–235, Los Alamitos, CA, USA, 1999. IEEE Computer Society Press.
- [13] R. Koschke. *Atomic Architectural Component Recovery for Program Understanding and Evolution*. Phd thesis, Fakultät Informatik, Elektrotechnik und Informationstechnik, Universität Stuttgart, Germany, August 2000.
- [14] R. Koschke and T. Eisenbarth. A framework for experimental evaluation of clustering techniques. In *IWPC 2000. 8th International Workshop on Program Comprehension*, pages 201–210, Limerick, Ireland, June 2000. IEEE.
- [15] R. Martin. OO Design Quality Metrics – An Analysis of Dependencies. Object-Oriented Programming Systems, Languages, and Applications (OOPSLA), October 1994.
- [16] B. Mitchell and S. Mancoridis. On the automatic modularization of software systems using the Bunch tool. *IEEE Transactions on Software Engineering*, 32(3):193–208, March 2006.
- [17] H. Muller, M. Orgun, S. Tilley, and J. Uhl. A reverse engineering approach to subsystem structure identification. *Journal of Software Maintenance: Research and Practice*, 5(4):181–204, December 1993.
- [18] G. J. Myers. *Reliable software through composite design*. Petrocelli/Charter, New York, 1975.
- [19] F. Plasil and S. Visnovsky. Behavior Protocols for Software Components. *IEEE Transactions on Software Engineering*, 28(11):1056–1076, 2002.
- [20] Prefuse. Prefuse Visualization Toolkits. <http://www.prefuse.org/>.
- [21] Recoder Team. Recoder homepage. <http://recoder.sourceforge.net/>. last retrieved 2007-10-25.
- [22] R. H. Reussner, S. Becker, H. Koziolok, J. Happe, M. Kuperberg, and K. Krogmann. The Palladio Component Model. Interner Bericht 2007-21, Universität Karlsruhe (TH), Faculty for Informatics, Germany, October 2007.
- [23] D. C. Schmidt, H. Rohnert, M. Stal, and D. Schultz. *Pattern-Oriented Software Architecture: Patterns for Concurrent and Networked Objects*. John Wiley & Sons, Inc., New York, NY, USA, 2000.
- [24] Software-Tomography GmbH. Sotograph homepage. <http://www.software-tomography.com/>. last retrieved 2007-10-30.
- [25] C. Szyperski, D. Gruntz, and S. Murer. *Component Software: Beyond Object-Oriented Programming*. ACM Press and Addison-Wesley, New York, NY, 2 edition, 2002.
- [26] M. Völter and T. Stahl. *Model-Driven Software Development*. Wiley & Sons, New York, NY, USA, 2006.
- [27] A. von Zitzewitz. Erosionen vorbeugen. *Javamagazin*, 2.07:28–33, Februar 2007.