Reverse Engineering Encapsulated Components from Object-Oriented Legacy Code

Rehman Arshad, Kung-Kiu Lau

rehman.arshad, kung-kiu.lau @manchester.ac.uk

School of Computer Science, University of Manchester

M13 9PL, United Kingdom

Abstract

Current component-directed reverse engineering approaches extract ADL-based components from legacy systems. ADL-based components need to be configured at code level for reuse, they cannot provide re-deposition after composition for future reuse and they cannot provide flexible re-usability as one has to bind all the ports in order to compose them. This paper proposes a solution to these issues by extracting X-MAN components from legacy systems. In this paper, we explain our component model and mapping from object-oriented code to X-MAN clusters using basic scenarios of our rule base.

Key Words —Reverse Engineering, Component Based Development[1]

1. Introduction

The term legacy systems usually refers to such software systems that are outdated, lack proper documentation and cannot support a new feature without breaking another logic yet they are vital to an organisation [5]. Unfortunately, most legacy code was designed with non-modular approach that cannot exploit the luxury of re-usability. For many companies, maintenance or comprehension of legacy code is crucial because some of their functions are too valuable to be discarded and too expensive to reproduce from scratch.

Component based development is a domain that revolves around the construction of systems from pre-built software units i.e., re-usability. Components extraction can reconstruct a legacy system as modular executable architectural units that can be reused across many systems. Component-directed[2] reverse engineering consists of following steps: 1) Capture the source code in appropriate notation (graph nodes etc.). 2) Define a rule base to map the extracted notation to abstraction model. 3) Formation of clusters i.e., code re-structuring 4) Mapping of clusters to a component model. Output of component-directed reverse engineering approaches is dependent on the definition of component each approach uses. Most approaches use loose definition of component. For them, a component is consisted of methods that belong together as they offer a specific functionality of the system. Such components can be giant classes, clusters or re-formation of the source code to get better cohesion and loose coupling. These approaches neither defines the extraction of explicit interfaces nor the composition mechanism of the extracted components (e.g., [9]). Such components are not feasible for reuse as non-explicit architecture cannot help in achieving a good re-usability.

Few like us, follow the szyperski's definition of components. This definition defines component as ”A unit of composition with contractually specified interfaces and explicit context dependencies only” [15]. These approaches extract explicit architecture (components with well-defined composition and interfaces).

Almost all the current reverse engineering approaches that extract explicit components are based on ADLs[3]. ADLs define required and provided services as ports (composition mechanism of ADLs). Ports use (indirect) method calls at code level to compose components together. ADL-based components have three major shortcomings from re-

[1] DOI reference number: 10.18293/SEKE2018-111

[2] Reverse Engineering that aims for the extraction of components.

[3] Components based on architecture description languages.

usability point of view: 1) Inability to select/de-select/alter ports without changing the code manually at all required places (for every single composition) to compose the components after retrieval. 2) One has to bind all the ports in order to reuse an ADL component i.e., non-flexible reusability. 3) It is impossible to re-deposit[4] a configured composition of components for reuse (e.g., composite component). Components have to be retrieved and configured as many times as the same composition is required. To the best of our knowledge, no such component-directed reverse engineering approach exists that can: do code-independent composition, allow to reuse the components without binding all services (ports) and support the re-deposition of composed components for further reuse or composition.

This paper presents a reverse engineering approach that can resolve the above stated issues. The mapping from extracted clusters to meta-model of our component model X-MAN [10] and working of our tool has already been explained in [4] (white boxes in Figure 1). In this paper, we explain how we: 1) Capture the object-oriented source code. 2) Map the captured notation to X-MAN clusters based on our rule base, by stating basic scenarios (red boxes in Figure 1). Section II of this paper compares our approach with other approaches that extract explicit architecture. Section III explains X-MAN component model. Section IV presents our approach using an example. Section V include conclusion and future work.

2. Related Work

There are quite a few approaches that follow szyperski's definition of components for reverse engineering.

JAVACompExt [3] is a heuristic based approach that extracts Abstract Data Type (ADT) components. The approach by Antoun *et al.* [1] re-engineers Java code into ArchJava components. Chouambe *et al.* [7] produces composite components from Java source code. Pattern-based Reverse Engineering of Design Components [12] extracts design components based on the structural descriptions of design patterns. A Reverse Engineering Approach to Subsystem Structure Identification [14] re-structures the system into a hierarchy of subsystems along with their high-level abstract representation as components. Washizaki [16] detects reusable part of object-oriented classes and transforms classes into JavaBeans components. *Archimetrix* [8] reconstructs the architecture in form of components from the source code after removing design deficiencies. Quality centric approach [11] focuses on quality of explicit interfaces by following a semantic-correctness model. Alshara *et al.* [2] extracts OSGi or SOFA components from object-oriented code. Components extraction in memory-constrained environments [16] identifies reusable part of an

[4]The term re-deposit-ability means ability to re-deposit the composed components after retrieval for future reuse.

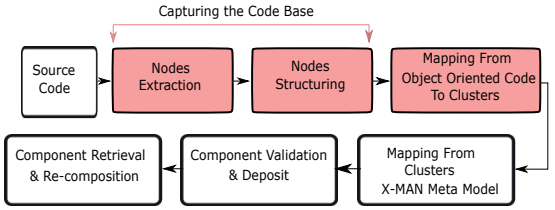


Figure 1: RX-MAN

Approach	Re-Composition	Repository Deposit	Componentization Indenpendency	Automated	Component Model
JAVACompExt	×	×	×	✓	UML
Antoun et al.	×	×	✓	×	ArchJava
Design Components	×	×	✓	✓	UML
Subsystem Structure Identification	×	×	✓	✓	ADL
Choumbe et al.	×	✓	×	✓	EJB
Washizaki	×	✓	✓	✓	JavaBeans
Archimetrix	×	×	✓	×	ADL
Alshara et al.	×	✓	✓	×	SOFA
Quality Centric	×	×	✓	✓	ADL
Memory Constrained..	×	✓	✓	✓	JavaBeans
RX-MAN	✓	✓	✓	✓	X-MAN

Table 1: Approaches based on Explicit Components

object oriented code and refactors the relative or surrounded code to reuse the identified part.

Few major shortcomings with these approaches are lack of automation, inability to retrieve from repository and inability to achieve code-independent composition of the extracted components. These approaches however, extract explicitly defined components with required and provided services. In Table 1, attribute *Repository Deposit* means whether an approach is based on a component model that supports repository or not. JAVACompext, Antoun's approach and Design Components are based on component models that do not support repository whereas, Subsystem Structure Identification, Archimetrix and Quality centric approach do not define or discuss the deposition of components via a repository. Lack of repository decreases reusability as components cannot be configured and preserved for retrieval. The attribute *Automated* shows whether an approach is automated or needs manual assistance. *Component Model* shows the component model that is followed for extraction of components. *Componentization independency* shows whether an approach is only applicable on source systems that are designed as separate packages.

Out of all the explicit approaches, our approach (that we call RX-MAN) is the only one that: supports component repository as part of its implementation, does not need code-level configurations for reuse, is automated, does not restrict to bind all the ports of a component being reused and supports composition of the re-deposited components. A well-known framework partially relevant to our approach is MoDisco [6]. MoDisco uses Architecture-driven modernization (ADM) to construct the Knowledge Discovery Meta-model (KDM). The core difference is that our approach aims for a specific meta-model (X-MAN) as a transformation model whereas, MoDisco aims for a cus-

tomised meta-models based on legacy technology and requirements[5].

3. X-MAN Component Model

Unlike ADL-based components, X-MAN component model is based on encapsulation i.e., an X-MAN component only has provided methods and no required ones. An atomic X-MAN component consists of a computation unit that has the implementation of methods and exposed functionality of specific methods (the methods that can be selected for composition). Methods are exposed as interfaces which are implemented in the computation unit. Any exposed method can be selected before instantiating a component and method's inputs and outputs can be used with the exposed methods of other X-MAN components. Computation only takes place in a computation unit, which is why this component model is encapsulated [13].

In case of a composite component, encapsulation is preserved by composition because a composite component consists of two or more atomic components composed together by composition connectors. Composition connectors in X-MAN are control structures that direct the route of execution. *Sequencer* (*SEQ*) composition connector provides sequencing of execution between two or more than two components and *Selector* (*SEL*) provides branching based on specific conditions[6]. If two components A and B will be instantiated with one exposed method each and composed by a sequencer, then there will be only two methods that will be involved in this composition. Basic semantics of X-MAN component model are shown in Figure 2 (lollipop in the Figure is a notation used to show the presence of exposed methods). One computation unit cannot interact with other units directly but only via composition connectors. Control of the components exists outside of computation units and that is why one does not need code-level configurations to reuse the components. Any component can be reused by composing it with others using appropriate composition connector [13] and exposed methods.

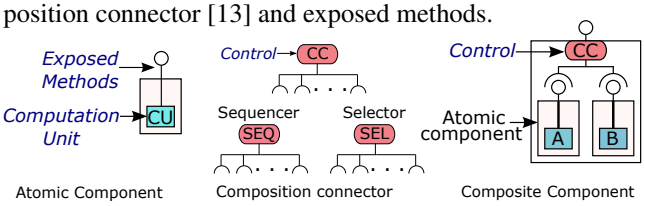


Figure 2: X-MAN Component Model

In ADL-based component models, control cannot be separated from computation and therefore, one needs code-level configurations to recompose required and provided services. X-MAN component model also supports redeposition of components after composition and composed integrations of components can be retrieved for future reuse. One does not need to bind all the ports at code level like ADLs but only need to select the exposed methods and appropriate composition connector for a composition.

[5]The implementation of our approach and MoDisco uses many common frameworks e.g., Ecore, eclipse modelling framework (EMF) etc.

[6]With *SEQ* (sequencing), *SEL* (branching) and *LOOP* (looping), X-MAN is Turing complete.

```
public class A{
public int provideSpeed(int speed)
{speed=100; this.returnSpeed(speed);}
public int returnSpeed(int topSpeed){this.saveInLog(topSpeed); return
    topSpeed;}
private void saveInLog(int value){ System.out.println("Value is
    saved");}}
```

Figure 3: Scenario 1

```
package com.A.scenario;
import java.util.*;
public interface A {
    public int provideSpeed (int speed);
    public int returnSpeed (int topSpeed);}

package com.A.scenario;
import java.util.*;
public class AImpl implements A {public AImpl() {}
    public int provideSpeed (int speed) {speed = 100; return speed;}
    public int returnSpeed (int topSpeed){this.saveInLog(topSpeed);
        return topSpeed;}
    private void saveInLog(int value){ System.out.println("Value is
        saved");}}
```

Figure 4: Mapped Code from Scenario 1

4. Our Approach: RX-MAN

This section uses an example of *Brake Control System* to demonstrate the code capturing and mapping of code from object-oriented classes to X-MAN clusters. Before showing the mapping in terms of an example, section below demonstrates few basic scenarios to show the rules of mapping.

4.1. Mapping from Source Code to X-MAN clusters

In RX-MAN, each input is a class, bunch of classes or a program (object-oriented code base). The input is mapped using a rule base against defined scenarios and output is one or more than one X-MAN components. The mapping is based on interactions and invocations of methods. Below are few basic scenarios to show the mapping rule base.

1) Single Class with no interaction: A single non-interactive class is the most trivial scenario in RX-MAN. In this scenario, all the methods that call each other belong to the same class. Output of this scenario would be one X-MAN component with all the methods of the class mapped to computation unit. Figure 3 is showing a non interactive single Java class. In Figure 3, methods *provideSpeed*

```
public class A{
    B obj= new B();
    public void provideSpeed(int speed){speed=100;
        obj.evaluateSpeed(speed);}}
public class B{
    public int evaluateSpeed(int topSpeed){int maxSpeed=200; return
        maxSpeed-topSpeed;}}
```

Figure 5: Scenario 2

```
public class A{
        B obj= new B();
        public void provideSpeed(int speed){speed=100;
              obj.evaluateSpeed(speed);}}
public class B{
        public int evaluateSpeed(int topSpeed){int maxSpeed=200; int
              recordSpeed=maxSpeed−topSpeed; this.saveInLog(recordSpeed);
              return recordSpeed;}
        private void saveInLog(int recordValue){system.out.println ("Value
              Logged Successfully"); }}
```

Figure 6: Scenario 3

and *returnSpeed* are marked as exposed methods. Method *saveInLog* is in the computation unit along with other two methods but it cannot be used as an exposed method because its modifier is private. The exposed methods of this component are mapped as an interface and computation unit has implementation of all the methods. Figure 4 shows the notation of mapped code of scenario 1 (inside X-MAN component). Figure 7(a) shows the notation of X-MAN component mapped from this scenario. Red boxes in Figure 7(a) shows the exposed functionality of this component i.e. exposed methods.

2) Two Classes with public-public methods interaction: Next possible scenario is the interaction of two Java classes in a code base. As our approach is based on interaction and invocation of methods, modifiers of methods play an important role in defining a scenario. Figure 5 is showing an example of two Java classes that interact with each other via methods with public modifiers. In this scenario, output would be just one X-MAN component. All the callers would be placed in one computation unit along with the methods they called. If a method M is in invocation list of more than one methods, it would be placed in the computation unit only once to avoid redundancy. This scenario assumes that all the interactions are between public methods and no method is neither invoking any private method nor dealing with any private class level variable. Figure 7(b) is showing the X-MAN component mapped for two Java classes of scenario 2[7].

3) Two Classes with private-public OR public-private methods interaction: This scenario has more possible outcomes than the previous two. If a private or a public method in Class A calls a public method in Class B, there are following possible scenarios.

1) Public method in Class B is neither accessing any private variable of the class nor it is calling any private method of B. In this case, such public method will be placed along with its caller in the same computation unit.

2) If method in Class B uses private variable of Class B or it calls some other private method of B, it cannot be simply placed with its caller. In this case:

a) If caller is private, the public method of B will be placed in both components (computation unit of A and computation unit of B as its dealing with private entities of both classes).

b) If caller is public, public method of B will only be part of computation unit of B. Its caller can access it using composition connector or by data input/output of an exposed method.

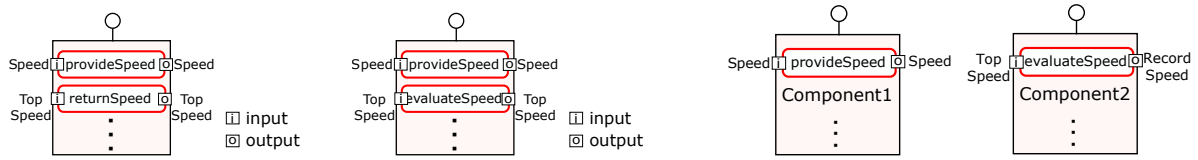
Figure 6 shows a scenario of public-private case. Method *provideSpeed* of Class A has method *evaluateSpeed* of Class B in its invocation list. Method *evaluateSpeed* is accessing private method *saveInLog* of Class B. Output of this scenario would be two X-MAN components. One component would have one method i.e. *provideSpeed*. The other component would have *evaluateSpeed* and *saveInLog* in its computation unit and *evaluateSpeed* would be the exposed method of second component. Figure 7(c) shows two components mapped from scenario 3. The purpose of explaining the above scenarios is to provide comprehension of the basic mapping mechanism. Clustering of methods based on their invocations and modifiers provide much better cohesion as only those methods would belong to same component that are associated and have loose coupling with rest of the components. To apply these scenarios on a full code base, one needs an appropriate notation that can capture the whole legacy code and preserve the relation and dependencies among all the entities. To capture the code base, our approach uses a customised parser that is written specifically for RX-MAN.

4.2. Capturing the Code Base

The customised parser used in this approach is based on Abstract Syntax Tree (AST) parser. The designed parser is more powerful than the default AST parser as it also extracts and maps invocation nodes from each method node in the code base. If a method A invokes method B, and method B invokes method C then our parser extracts and connects all nodes of the method C to method A as both are indirectly connected by method B. AST parser extracts one big tree of nodes from a code base in which all the nodes are connected hierarchically e.g., starting node would be compilation unit (class level or package level) connected with its sub nodes i.e., class declarations, class variables etc. Each class declaration node is further connected to its method nodes and each method node is connected with its sub nodes. This hierarchy of nodes goes till the last level which is simple name nodes i.e., name of local variables etc.

It is impossible to trace and cluster the chain of all possible method interactions and invocations from this one big complex tree. Therefore, RX-MAN parser indexes each method of the code base and connect all associated nodes with every method. Figure 8 is showing the extraction of nodes using RX-MAN parser. Each method node index has information about its parent class, parent package and class variable this method uses. Along with this information, each method node index is connected with all the method

[7]In case of void methods, output of an exposed method is boolean that indicates termination of execution of that method



(a) X-MAN Component mapped from Scenario 1 (b) X-MAN Component mapped from Scenario 2 (c) X-MAN Components mapped from Scenario 3

Figure 7: X-MAN: Components Mapped From Scenarios

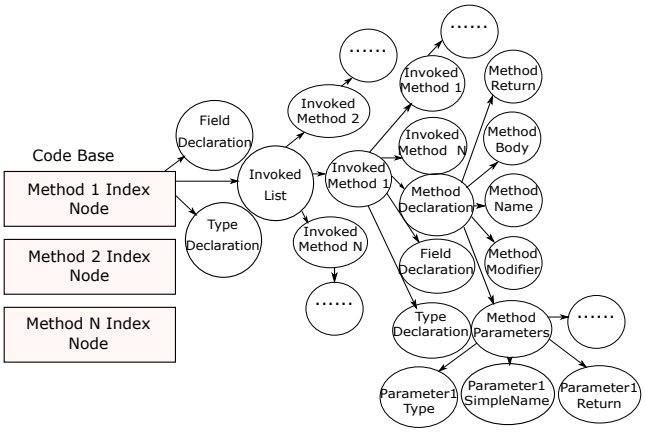


Figure 8: RX-MAN Parser

```
package vehicle.control.speedcontrol;
import vehicle.brakecontrol.BrakeControl;
public class SpeedMonitoring {
BrakeControl obj;
public SpeedMonitoring(){}
public double collisionTimeCalculator (double speed, double distance)
{speedMonitoringValue(distance/speed);
return distance/speed;}
public boolean speedMonitoringValue(double collisionEstimate)
{boolean time=false;
if (collisionEstimate <15)
{time=true; obj.collisionParametersActivation (time);}
else {obj.collisionParametersActivation (time);} return time;}}

package vehicle.brakecontrol;
public class BrakeControl {
public BrakeControl(){}
public boolean collisionParametersActivation (boolean flag){
if (flag==true){BrakeSystemActivation(flag);}
else{TimeTriggerValue(flag);} return flag;}
private void BrakeSystemActivation(boolean value){
System.out.println ("Brakes Applied");}
public boolean TimeTriggerValue(boolean value){return value;}
```

Figure 9: Brake Control System

it invokes directly or indirectly. This mapping makes sure that no indirect invocation goes undetected. In short, starting from each method in a code base, each method node index is connected with whole chain of invocations it causes in a code base (Figure 8). Therefore, each cluster of RX-MAN is consisted of restructured associated nodes based on rules of method's interactions and invocations.

4.3. Example: Brake Control System

Fig 9 shows a simple example of brake control system that is reverse engineered using our approach. In the given example, there are two classes. Class *SpeedMonitoring* has methods *collisionTimeCalculator* (for calculating time till collision) and *speedMonitoringValue* (for automatic brake

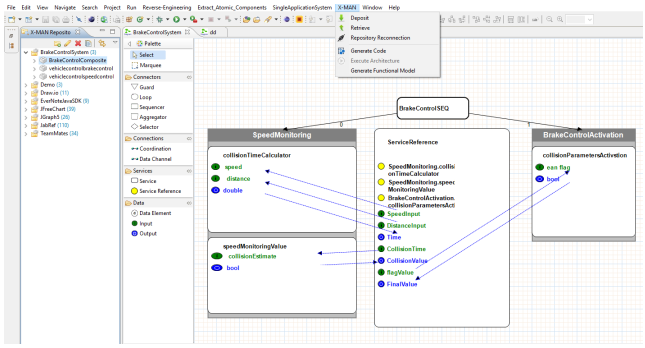


Figure 10: Deposition of A Composite Component

mode if time till collision is less than 15 seconds). Method *speedMonitoringValue* invokes *collisionParametersActivation* from Class *BrakeControl*. Depending on the value of time, method *collisionParametersActivation* either invokes *BrakeSystemActivation* or *TimeTriggerValue*.

According to our approach, method *speedMonitoringValue* has one method node against its invocation node i.e., *collisionParametersActivation* and method *collisionParametersActivation* has two method nodes against its invocation node i.e., *BrakeSystemActivation* and *TimeTriggerValue* (hence two indirect invocation nodes against *speedMonitoringValue*). As the method *BrakeSystemActivation* is private (scenario 3) therefore, the approach will map the whole code to two clusters.

RX-MAN tool maps these clusters to X-MAN metamodel and extracts two components. First cluster has methods *speedMonitoringValue* and *collisionTimeCalculator* (both will be mapped as exposed methods of an X-MAN component). Second cluster has methods *collisionParametersActivation*, *BrakeSystemActivation* and *TimeTriggerValue* (from this cluster method *BrakeSystemActivation* cannot be mapped as an exposed method as it is private.

Fig 10 is showing a possible case of composition of RX-MAN using a composition connector sequencer (SEQ). *SpeedMonitoring* component (extracted from first cluster) will be triggered first as this route has 0 (lower number means higher priority) and component *BrakeControlActivation* (extracted from second cluster) will be triggered after that. It is one valid case of composition as the component *BrakeControlActivation* will perform its execution after getting *collisonValue* from component *SpeedMonitoring*. Fig 10 is also showing that this composite component has been deposited in the *BrakeControlSystem* (X-MAN repository at left) and can be instantiated in future to be reused or re-

Code Base	Java Classes	X-MAN Components	Abstraction Ratio	Total Methods	Processing Time of RX-MAN
Draw.io	**101**	**11**	**9%**	**892**	**06 Secs**
EverNote SDK	**106**	**09**	**12%**	**3910**	**08 Mins, 11 Secs**
JabRef	**935**	**110**	**8.5%**	**6434**	**06 Mins, 28 Secs**
JFree Chart	**993**	**39**	**25.46%**	**10274**	**20 Mins, 43 Secs**
JGraph5	**171**	**26**	**7%**	**3482**	**39 Secs**
TeamMates	**815**	**34**	**23.97%**	**7519**	**05 Mins, 76 Secs**

Figure 11: Evaluation Cases

composed further.

The proposed approach has been applied on six legacy code basis, available for empirical evaluation at *GitHub* and *Quality Corpus*. Figure 11[8] is showing the summarisation of results, obtained by our approach.

5. Discussion and Conclusion

This paper presents two important steps of our approach: code capturing and mapping from object-oriented code to X-MAN clusters. We also demonstrated an example of Brake Control System and show a valid case of composition in our tool.

The biggest threat to validity of RX-MAN is the lack of consideration to important relations in an object-oriented language e.g., aggregation, composition and inheritance etc. These relations, if mapped, can provide much better cohesion and hence better re-usability. Future work includes expanding this approach beyond interactions of methods to map control statements in the code (*if, switch, loops etc.*) to composition connectors of X-MAN. To the best of our knowledge, ours is the only approach that can reuse and compose the extracted components without any code-level configurations with ability of re-deposition of components.

References

[1] Marwan Abi-Antoun, Jonathan Aldrich, and Wesley Coelho. A case study in re-engineering to enforce architectural control flow and data sharing. *Journal of Systems and Software*, 80(2):240–264, 2007.

[2] Zakarea Al-Shara, Abdelhak-Djamel Seriai, Chouki Tibermacine, Hinde Lilia Bouziane, Christophe Dony, and Anas Shatnawi. Materializing architecture recovered from oo source code in component-based languages. In *ECSA: European Conference on Software Architecture*, 2016.

[3] Nicolas Anquetil, Jean-Claude Royer, Pascal Andre, Gilles Ardourel, Petr Hnetynka, Tomas Poch, Dragos Petrascu, and Vladiela Petrascu. Javacompext: Extracting architectural elements from java source code. In *Reverse Engineering, 2009. WCRE'09. 16th Working Conference on*, pages 317–318. IEEE, 2009.

[4] R. Arshad and K.-K. Lau. Extracting executable architecture from legacy code using static reverse engineering. In *Proceedings of Twelfth International Conference on Software Engineering Advances*, pages 55–59. IARIA, 2017.

[5] K. Bennett. Legacy systems: coping with success. *IEEE Software*, 12(1):19–23, Jan 1995.

[6] Hugo Bruneliere, Jordi Cabot, Grégoire Dupé, and Frédéric Madiot. Modisco: A model driven reverse engineering framework. *Information and Software Technology*, 56(8):1012–1032, 2014.

[7] Landry Chouambe, Benjamin Klatt, and Klaus Krogmann. Reverse engineering software-models of component-based systems. In *Software Maintenance and Reengineering, 2008. CSMR 2008. 12th European Conference on*, pages 93–102. IEEE, 2008.

[8] Markus Detten, Marie Christin Platenius, and Steffen Becker. Reengineering component-based software systems with archimetrix. *Softw. Syst. Model.*, 13(4):1239–1268, October 2014.

[9] J. M. Favre, F. Duclos, J. Estublier, R. Sanlaville, and J. J. Auffret. Reverse engineering a large component-based software product. In *Proceedings Fifth European Conference on Software Maintenance and Reengineering*, pages 95–104, 2001.

[10] Nannan He, Daniel Kroening, Thomas Wahl, Kung-Kiu Lau, Faris Taweel, C Tran, Philipp Rümmer, and S Sharma. Component-based design and verification in X-MAN. *Proc. Embedded Real Time Software and Systems*, 2012.

[11] S. Kebir, A. D. Seriai, S. Chardigny, and A. Chaoui. Quality-centric approach for software component identification from object-oriented code. In *2012 Joint Working IEEE/IFIP Conference on Software Architecture and European Conference on Software Architecture*, pages 181–190, Aug 2012.

[12] Rudolf K Keller, Reinhard Schauer, Sébastien Robitaille, and Patrick Pagé. Pattern-based reverse-engineering of design components. In *Proceedings of the 21st international conference on Software engineering*, pages 226–235. ACM, 1999.

[13] Lau Kung-kiu et al. *An Introduction To Component-based Software Development*, volume 3. World Scientific, 2017.

[14] Hausi A Müller, Mehmet A Orgun, Scott R Tilley, and James S Uhl. A reverse-engineering approach to subsystem structure identification. *Journal of Software: Evolution and Process*, 5(4):181–204, 1993.

[15] Clemens Szyperski. *Component software: beyond object-oriented programming*. Pearson Education, 2002.

[16] Hironori Washizaki and Yoshiaki Fukazawa. Extracting components from object-oriented programs for reuse in memory-constrained environments.

[8] Abstraction Ratio means average size of components in terms of code classes