Reverse Engineering Encapsulated Components from Object-Oriented Legacy Code

DOI:
10.18293/seke2018-111

Document Version
Accepted author manuscript

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-based reverse engineering approaches usually extract ADL-based components from legacy systems. ADL-based components need to be configured at code level for reuse, they cannot provide re-deposition of composed components 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, Static Analysis, Component Based Development

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The term legacy systems usually refers to such software systems that are outdated, lack proper documentation and cannot support a new feature without breaking another logic yet they are vital to an organisation [9]. 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. Studies show that 50-60 percent of software engineering effort is spent trying to understand source code [8].

Component based development is a domain that revolves around the construction of systems from pre-built software units i.e., re-usability. Instead of extracting semantics, like general reverse engineering for analysing a system, components extraction can reconstruct a legacy system as modular executable architectural units that can be reused across many systems. Component based reverse engineering consists of following steps: 1) Capture the source code in appropriate notation (can be graph nodes or source code metrics). 2) Define a rule base to map the extracted notation to abstraction model. 3) Formation of clusters from abstraction model, based on algorithms 4) Map the clusters to output notation (semantics of the selected component model). Output of component based reverse engineering 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., [13]). Such components are not feasible for reuse as lack of comprehension of legacy systems and 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” [20]. These approaches like us, 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[1] (e.g.,[3]). ADLs define required and provided services as ports (composition mechanism of ADLs). Ports use (indirect) method calls at code level [6, 7] to compose components together. ADL-based components have three major shortcomings from re-usability point of view: 1) In ADL-based components, one cannot select/de-select/alter ports without changing the code manually at all required places to re-compose the components after retrieval. Each different integration of components demands different configuration to reuse the components. Especially, in case of legacy systems, where adequate comprehension of implementation is not expected, re-configuration at code level is much more complex and time-consuming. 2) ADL-based components can only be retrieved to use as is. One has to bind all the ports in order to use an ADL component. Therefore, ADL-based components provide non-flexible re-usability. 3) It is impossible to re-deposit[2] a configured integration of components for reuse (e.g., composite component). Components have to be retrieved and configured as many times as the same integration is required for every reuse in different systems.

To the best of our knowledge, no such component based reverse engineering approach exists that can:
1) Re-compose the reverse engineered components without changing the code at all required places.
2) Allow to reuse the components without binding all required and provided services.
3) Support the re-deposition of re-composed components for future reuse of the same integrated configuration.

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 [14] and working of our tool has already been explained in [4][5] (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 purpose

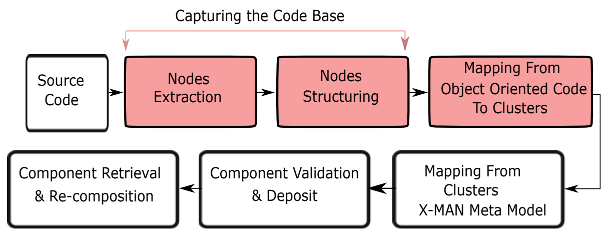

Capturing the Code Base
Source Code
Nodes Extraction
Nodes Structuring
Mapping From Object Oriented Code To Clusters
Mapping From Clusters X-MAN Meta Model
Component Validation & Deposit
Component Retrieval & Re-composition

Figure 1: RX-MAN

of this research is to avoid the system corrosion by making architecture explicit in the source code. The approach by Antoun *et al.* [1] re-engineers Java code into ArchJava components [2]. Chouambe *et al.* [10] produces composite components from Java source code. Pattern-based Reverse Engineering of Design Components [16] extracts design components based on the structural descriptions of design patterns. A Reverse Engineering Approach to Subsystem Structure Identification [19] is another approach that re-structures the system into a hierarchy of subsystems along with their high-level abstract representation as components. Washizaki [21] detects reusable part of object-oriented classes and transforms classes into JavaBeans components automatically. *Archimetrix* [11] is an approach that can reconstruct architecture in form of components from the source code after removing design deficiencies. Quality centric approach [15] focuses on quality of explicit interfaces by following a semantic-correctness model. Components extraction in memory-constrained environments [21] is an approach that identifies reusable part of an object oriented code and refactor the relative or surrounded code to reuse the identified part. This approach follows the JavaBeans component model [12]. *L2CBD* [17] stands for legacy systems to component based development and this approach is different from others because it is a methodology rather than a concrete approach itself. Any proposed approach can use L2CBD methodology to transform legacy code into components. This methodology consists of planning phase, re-engineering phase, componentization phase and component testing.

Few major shortcomings with these approaches are lack of automation, inability to retrieve from repository and inability to achieve re-composition or code independent re-usability of the extracted components after retrieval. These approaches however, extract components with defined required and provided services based on the semantics of a component model.

All the above approaches have same predefined order of steps. Every approach starts with capturing the code base in some notation. Based on a set of rules, formation of clusters is the second step (by using graph dependencies, directed graphs, code metrics etc.). The last step is the mapping from extracted structure to semantics of a component model. In Table 1, attribute *Repository Deposit* means whether an ap-

[1]Components based on architecture description languages
[2]The term re-deposit-ability means ability to re-deposit the composed components after retrieval for future reuse.

Approach	Re-Composition	Repository Deposit	Componentization Indenpendency	Automated	Component Model
JAVACompExt	x	x	x	✓	UML
Antoun et al.	x	x	✓	x	ArchJava
Design Components	x	x	✓	✓	UML
Subsystem Structure Identification	x	x	✓	✓	ADL
Choumbe et al.	x	✓	x	✓	EJB
Washizaki	x	✓	✓	✓	JavaBeans
Archimetrix	x	x	✓	x	ADL
L2CBD	x	x	✓	x	Not Defined
Quality Centric	x	x	✓	✓	ADL
Memory Constrained..	x	✓	✓	✓	JavaBeans
RX-MAN	✓	✓	✓	✓	X-MAN

Table 1: Approaches based on Explicit Architecture (Components)

proach 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, L2BCD and Quality centric approach do not define or discuss the deposition of components via a repository. Lack of repository decreases re-usability as components cannot be configured and preserved for future 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 re-composition of extracted components.

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 component consists of a computation unit that has the implementation of methods, exposed functionality of specific methods (services that are used to send/receive data elements) and an invocation connector. Methods are exposed as services by invocation connector (lollipop in Figure 2). Computation only takes place in a computation unit, which is why this component model is encapsulated [18].

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. All such atomic components can only provide methods by their invocation con-

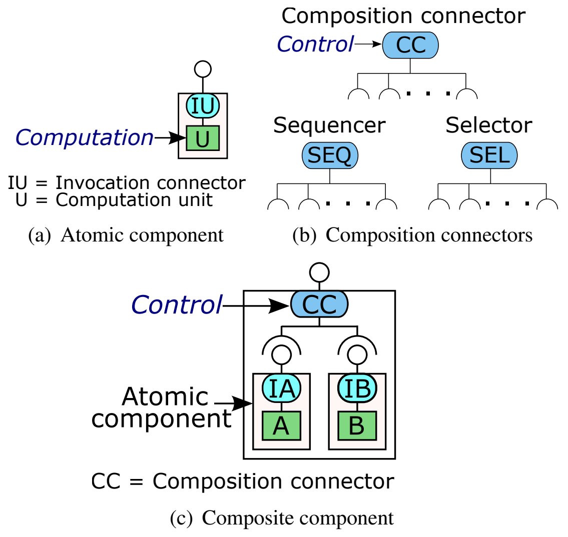


Figure 2: X-MAN: Components and composition connectors.

```
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: Single Non-Interactive Class

nectors and co-ordinate with one another using 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[3]. Basic semantics of X-MAN component model are shown in Figure 2. One computation unit cannot interact with other 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[4] (control mechanism does not exist at code level). Any component can be reused by integrating it with others using appropriate composition connectors [18].

In ADL-based component models, control cannot be separated from computation and therefore, one needs code-level configurations to recompose required and provided services (control mechanism i.e., ports exist at code level). X-MAN component model also supports re-deposition of components after composition and composed integrations

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

[4]Encapsulation in X-MAN does not mean code hiding, it means no computation goes outside of computation unit.

```
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

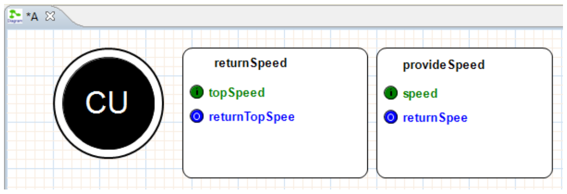


Figure 5: Atomic Component in X-MAN Tool

of components can be retrieved for future reuse [5]. One does not need to bind all the provided methods at code level like ports but only need to use services and compose components by appropriate composition connectors.

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. We used the same example in [5] to demonstrate the working of our tool and repository deposition (and re-deposition after composition). 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 Object-Oriented 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.

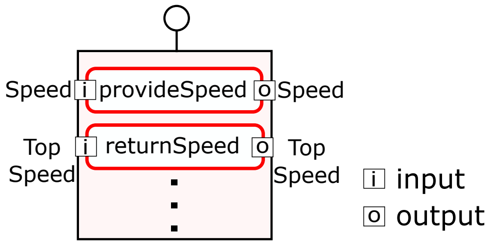


Figure 6: X-MAN component mapped from Scenario 1

```
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 7: Scenario 2: Two classes with public-public interactions

- ***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. Only public methods of the class are flagged as possible services. Figure 3 is showing a non interactive single Java class. In Figure 3, methods *provideSpeed* and *returnSpeed* are marked as services. Method *saveInLog* is in computation unit along with other two methods but it cannot be a service because its modifier is private. The services 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) and Figure 5 shows the extracted atomic component in our tool. Figure 6 shows the notation of X-MAN component mapped from this scenario. Red boxes in Figure 6 shows the exposed functionality of this component i.e. services. This atomic component can be composed with others via composition connectors and can provide methods via invocation connector.
- ***Two Classes with public-public methods interaction***: Next possible scenario is the interaction of two Java

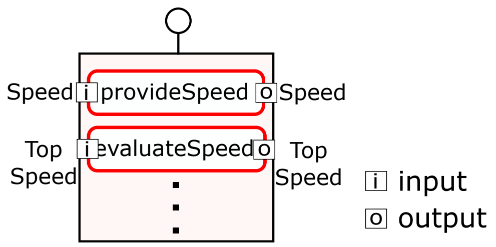


Figure 8: X-MAN component mapped from Scenario 2

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 7 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 8 is showing the X-MAN component mapped for two Java classes of scenario 2[5].

- ***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 service (for data input/output).

  Figure 9 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 service of second component. Figure 10 shows two components mapped from scenario 3.

[5]In case of void methods, output of service is boolean that indicates termination of execution of the service

```
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 9: Scenario 3: Two classes with private-public OR public private interactions

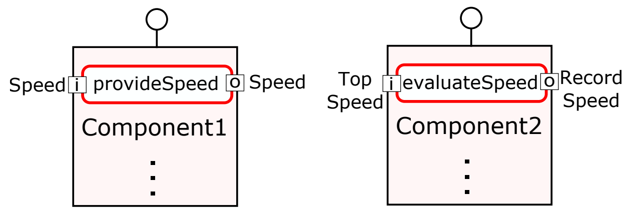


Figure 10: X-MAN components mapped from Scenario 3

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. Of-course, in case of legacy code basis, the scenarios would become more and more complex and number of components are dependent on interactions of methods. We have presented our algorithms in [4, 5] that cover more complex scenarios. To apply these scenarios on a full code base, one needs an appropriate notation that can capture the whole legacy code and preserves 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 indi-

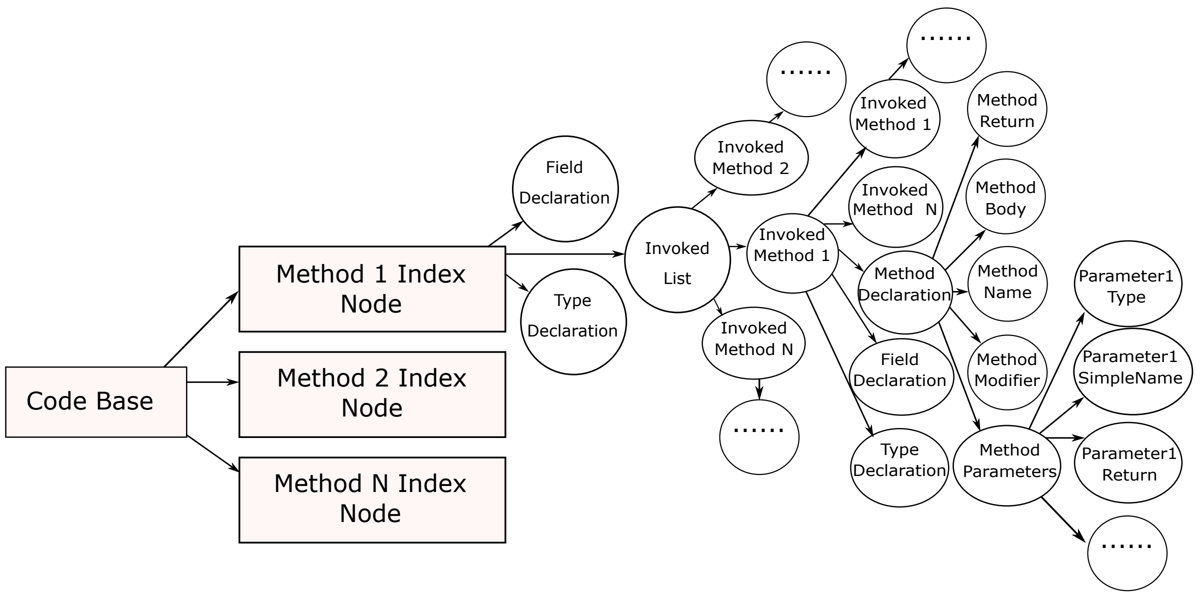
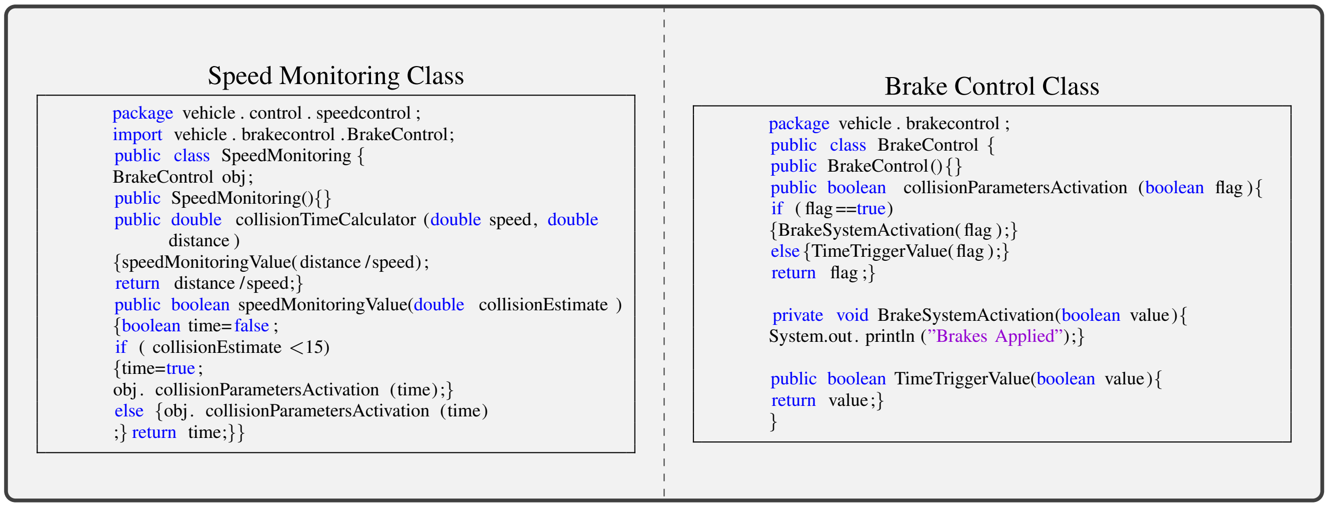


Figure 11: RX-MAN Parser



Speed Monitoring Class

```
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;}}
```

Brake Control Class

```
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 12: Brake Control System

rectly 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 i.e., method parameters nodes, method return node, method body node etc. 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 11 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 it invokes directly or indirectly. This mapping makes sure that no indirect invocation goes undetected. This kind of mapping can cause duplication of methods in a cluster as many methods may invoke same set of methods indirectly, but such redundancy can be removed before submitting the mapped components into repository. In short, starting from

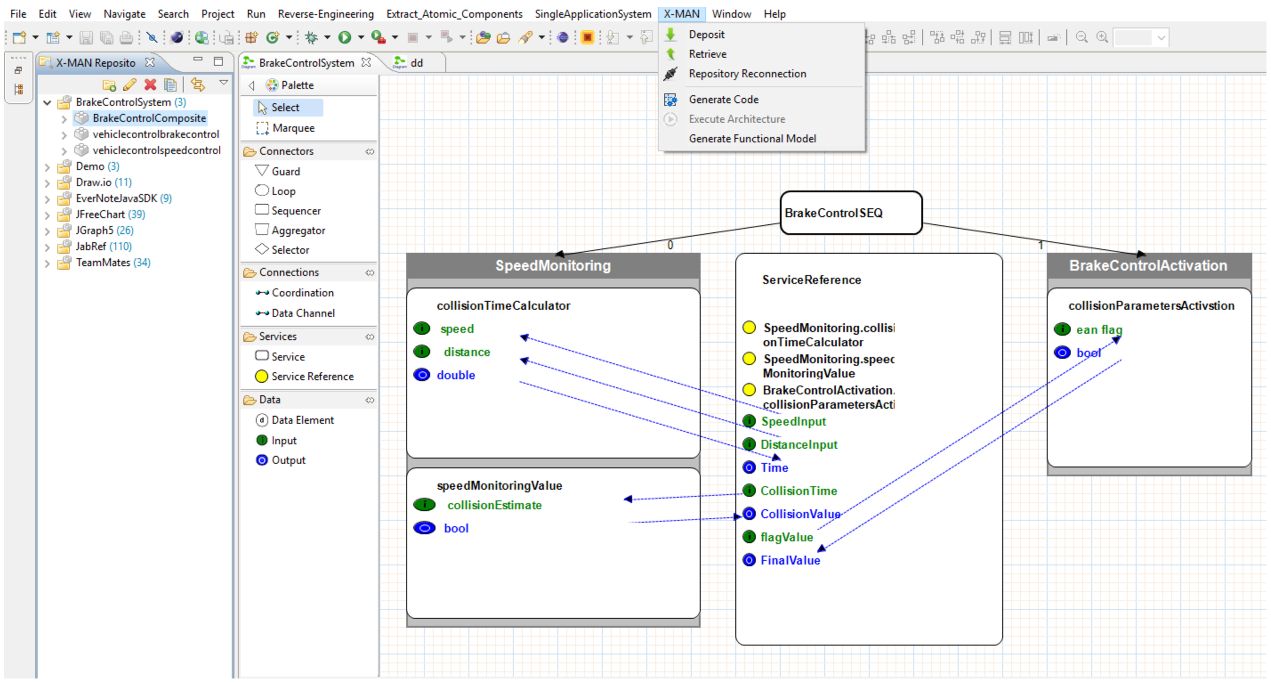


Figure 13: Deposition of A Composite Component after Re-composition

each method in a code base, each method node index is connected with whole chain of invocations it causes in a code base (Figure 11). 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 12 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 by using speed of the vehicle and distance from the next vehicle) and *speedMonitoringValue* (for automatic brake mode if time till collision is less than 15 seconds). Method *speedMonitoringValue* invokes *collisionParametersActivation* from Class *BrakeControl* that belongs to a different class. 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* via *collisionParametersActivation*). 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 meta-model and extracts two components [6]. First cluster has methods *speedMonitoringValue* and *collisionTimeCalculator* (both will be mapped as services of an X-MAN component as these methods are public and do not access any private variable/method directly). Second cluster has methods *collisionParametersActivation*, *BrakeSystemActivation* and *TimeTriggerValue* (from this cluster method *BrakeSystemActivation* cannot be mapped as a service as it is private.

Fig 13 is showing a possible case of re-composition of RX-MAN. Both the extracted components are composed using a composition connector (Sequencer). *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 re-composition as the component *BrakeControlActivation* will perform its execution after getting *collisonValue* from component *SpeedMonitoring*. Fig 13 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-composed further. We do not need any manual code changes due to composition connectors of X-MAN and we can also generate code of this composite component using RX-MAN tool.

[6]The steps of mapping of clusters to X-MAN meta-model and details about the tool have been presented in [5]

5. Discussion and Conclusion

This paper presents two important steps of our approach of reverse engineering: 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 re-composition in our tool. We have already applied our approach to six large legacy code basis and all the details of evaluation have been presented in [5].

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 in the extracted components and hence better reusability. Future work includes expanding this approach beyond methods'interactions 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 component based reverse engineering approach that can reuse and re-compose the extracted components without any code-level configurations and supports the repository and 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] Jonathan Aldrich, Craig Chambers, and David Notkin. Archjava: connecting software architecture to implementation. In *Proceedings of the 24th international conference on Software engineering*, pages 187–197. ACM, 2002.

[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] R. Arshad and K.-K. Lau. Reverse engineering recomposable components from legacy code. In *Proceedings of 7th International Conference on Software and Computing Technologies*. LNSE, 2018.

[6] Timo Asikainen, Timo Soininen, and Tomi Männistö. A Koala-Based Approach for Modelling and Deploying Configurable Software Product Families. In *Software Product-Family Engineering*, pages 225–249. Springer, 2004.

[7] Rabih Bashroush, T John Brown, Ivor Spence, and Peter Kilpatrick. Adlars: An architecture description language for software product lines. In *Software Engineering Workshop, 2005. 29th Annual IEEE/NASA*, pages 163–173. IEEE, 2005.

[8] Victor R Basili. Evolving and packaging reading technologies. *Journal of Systems and Software*, 38(1):3–12, 1997.

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

[10] 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.

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

[12] Wolfgang Emmerich and Nima Kaveh. Component technologies: Java beans, com, corba, rmi, ejb and the corba component model. In *ACM SIGSOFT Software Engineering Notes*, volume 26, pages 311–312. ACM, 2001.

[13] 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.

[14] 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.

[15] 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.

[16] 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.

[17] Haeng-Kon Kim and Youn-Ky Chung. Transforming a legacy system into components. In Marina Gavrilova, Osvaldo Gervasi, Vipin Kumar, C. J. Kenneth Tan, David Taniar, Antonio Laganá, Youngsong Mun, and Hyunseung Choo, editors, *Computational Science and Its Applications - ICCSA 2006*, pages 198–205, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

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

[19] 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.

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

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