Identifying Components in Object-Oriented Programs using Dynamic Analysis and Clustering

Simon Allier
DIRO
Université de Montréal
Université de Bretagne-Sud

Houari A. Sahraoui
DIRO
Université de Montréal

Salah Sadou
VALORIA
Université de Bretagne-Sud

Abstract

We propose an approach for component candidate identification as a first step towards the extraction of component-based architectures from object-oriented programs. Our approach uses as input dynamic call graphs, built from execution traces corresponding to use cases. This allows to better capture the functional dependencies between classes. The component identification is treated as a clustering problem. To this end, we use formal concept analysis and design heuristics.

We evaluate the feasibility of our approach on two programs. The obtained results are very satisfactory from both the performance and qualitative points of view.

1 Introduction

The component paradigm proposes to build systems from slightly coupled elements, called components. Each component implements a particular function of the system. Components communicate according to sets of required and provided interfaces. These characteristics give the component paradigm better abstraction capabilities, more flexibility for evolution and maintenance, and a better reusability in comparison to the object-oriented (OO) paradigm. For these reasons, it is interesting to extract component-based architectures from an object oriented systems. Such architectures allow better understanding of legacy OO code as stated in [11]. Moreover, identified components can be packaged and integrated into component libraries to be reused in new applications [21, 6].

As will be discussed in section 6, some contributions have been proposed to extract component-based architectures from OO code. The component identification process uses dependencies extracted by static analysis to group classes into components. Such groupings produce, in our opinion, architectures that represents the development viewpoint and not the functional/logical viewpoint that should characterize component-based systems.

To address the issue of relating dependencies between classes to program functions, we propose an approach that uses execution traces. These are obtained by running the program on a set of use cases corresponding to its associated functions. To identify the component candidates, we defined an automated process, based on formal concept analysis, that groups classes into components using dynamic call graphs derived from the execution traces. Our aim is not to completely construct the components. We rather identify, on the one hand, the classes that may constitute a component and, on the other hand, its provided and required interfaces. Thus, based on a consensual definition of software components, our approach is an aid for restructuring an OO architecture into a component-oriented one. This approach has two advantages: i) The restructuring is done at the architectural level (high level of abstraction); ii) It can be used for any concrete component model, since these models share the same definition of software component.

We evaluated our approach, in a first step, with a case study on a relatively small JAVA program (39 classes). Six components were identified, and correspond to exiting functions. In a second step and to assess the scalability of the proposed approach, we applied it on a bigger system (204 classes). This last study showed that the results from the performance and quality point of views are comparable to those of the first study.

The remainder of this paper is organized as follows. Section 2 introduces some definitions, highlights the component identification issues, and gives an overview of our approach. The extraction of dynamic call graphs is described in Section 3. The details of the identification process are given in Section 4. Our approach is evaluated and discussed in Section 5. Section 6 introduces the related work. Concluding remarks are given in Section 7.

2 Background and Approach Overview

Before describing the addressed issues (Section 2.2) and the contributions (Section 2.3), we start by giving the definitions of some main concepts involved in this work.

2.1 Definitions

The goal of our work is to identify component candidates with the perspective of program understanding and/or code reuse. The main concept manipulated is then component.

A software **component** is a system element offering a predefined service or event, and able to communicate with other components. When considering our problem, components are not designed as such. They are retrieved by analyzing OO code. From the detection perspective, a component is a set of classes performing a specified function. Finding components is a first step towards architecture recovery. An **architecture** is the fundamental organization of a system embodied in its components, their relationships to each other, and to the environment, and the principles guiding its design and evolution [10]. This is a general definition of an architecture. Architecture can be depicted using different viewpoints [7, 10] as could be a building architecture (floor plans, plumbing, electrical wiring, etc.). Two viewpoints are of interest for our work: functional/logical and development. We define these two viewpoints according to Kruchten's 4+1 view model [7]. The **logical architecture** primarily supports the functional requirements, *i.e.,* what the system should provide in terms of services to its users. The **development architecture** focuses on the actual software module organization of the software development environment. The logical architecture can be viewed as a set of components representing system functions where the development architecture is defined in terms of modules. In some cases, components are physically defined as modules. In these case the logical architecture is used to define the development architecture. In many other situations logical and development viewpoints are orthogonal.

As explained in the following sections, dynamic analysis is used to capture the functional interactions between classes. To ease the understanding of our approach, we define the used concepts: use cases, use case scenario, execution trace, call graph, static call graph, and dynamic call graph. A **use case** describes a function provided by a system that yields a visible result for an actor. A concrete instance of a use case that captures a specific interaction sequence is called **scenario**. In our context, we define an **execution trace** as the sequence of method calls corresponding to the execution of a system according to a use case scenario. Interactions between methods, extracted by static or dynamic analysis define a call graph. A **call graph** is a directed graph that represents calling relationships between methods in a program. Attribute accesses are also considered. We can also represent the call graph between classes by replacing method/attribute nodes by corresponding-class nodes. A **static call graph** is a call graph where calling/accessing relationships are extracted by static analysis. In other words, it contains all the methods/atributes that can be potentially called/accessed and the methods that call/access them. A **dynamic call graph** is a call graph where calling/accessing relationships are extracted by analyzing execution traces. In other words, it contains all the methods/atributes that are actually called/accessed in at least one execution trace, the methods that call/access them, and possibly the number of occurrences of each relationship.

2.2 Identification issues

Different criteria may serve to group classes into modules corresponding to different types of cohesion [14]. Consequently, in the architecture recovery/optimization perspective, the choice of information that can be exploited depends on the kind of the targeted architecture (viewpoint). As it will be discussed in Section 6, dependencies extracted by static analysis (structural dependencies between classes) are often used. But, as it is stated in [9], "*A good object-oriented design does not necessarily make a good component-based design, and vice versa.*". Indeed, the structural relationships (static dependencies) between elements of an OO source code reflect its OO design. Trying to extract automatically a component structure by analyzing statically the OO source code, *i.e.,* using its static dependencies, leads to a solution that conforms to OO design.

To avoid this problem, we propose to go back to a step where OO concepts do not influence the architectural design and to the main component design principle: composition is guided by the functional requirements. Consequently, our approach starts from the use case documentation to perform the component identification. Execution traces obtained by running the program on a set of use cases allow us to identify the dependencies between classes (methods calls) that contribute to perform a particular function of the system. We view then the component identification as the grouping of classes whose interactions conform to use case-based dependencies. Moreover, provided and required interfaces of such groups of classes can be extracted by focusing on external dependencies, which allows, once restructured by the designer, to package actual components.

2.3 Approach Overview

In our approach, a component is seen as a set of classes collaborating to perform a particular function. The provided and required interfaces are represented by the method calls between classes belonging to different components. The identification of component candidates is then defined as the partition of the set of classes of the considered program into groups. A candidate (group of classes) is determined by both the cohesion (internal dependencies) and the coupling (external dependencies). In other words, considering the method calls, a group of classes must exhibit a high cohesion, *i.e.,* a good connectivity between the included classes, and low coupling with classes from other groups.

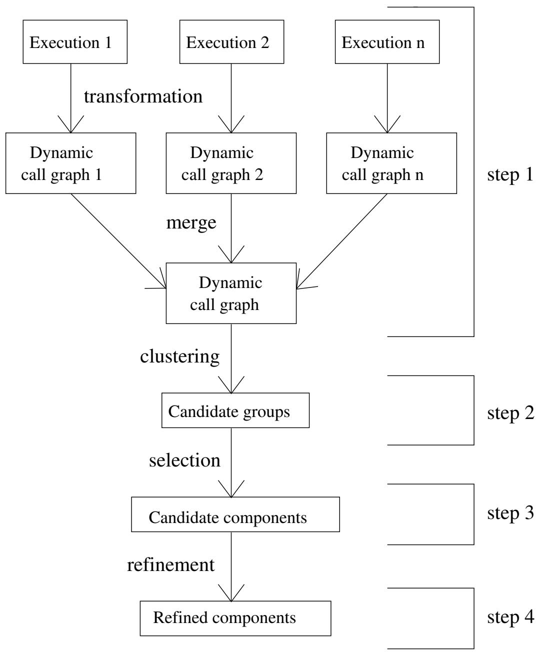

Figure 1: The four steps of component identification

Our approach defines the four following steps (Figure 1): (1) data extraction, (2) possible class groups identification, (3) candidate component selection, and (4) candidate component refinement. Data (method calls) are extracted using dynamic analysis. They are obtained by executing typical use cases of the program and by grouping the corresponding execution traces into what we call a dynamic call graph (DCG). One of the hypotheses of our work is that use cases can be derived from the application documentation and/or test cases. Using the DCG, a concept lattice is built. The lattice's nodes define the space of all the meaningful groups of classes, *i.e.,* groups of interrelated classes. From this set, a subset is selected according to a function that measures the quality of groups, and by considering the formal properties of set partitions. The selected groups are then optimized using a set of heuristics. The resulting set of candidate components and their connections could serve as a basis for the creation of the component-based architecture.

The first step of our approach is described in Section 3. The three other steps are discussed in detail in Section 4.

3 Dynamic-Call-Graph Extraction

We start from the hypothesis that a set of classes which "strongly" collaborate to perform a set of functions can be grouped into a component. Therefore, the candidate component identification can be viewed as a graph partioning problem. To focus on functional dependencies, we decided to use an abstraction called Dynamic Call Graph (DCG) as input for our identification process.

We define a dynamic call graph as a directed graph $G(V, E)$ where V is the set of nodes and $E \in V \times V$ is the set of edges between these nodes. Nodes of the graph represent the classes and edges represent the method calls between classes. In fact, in addition to method calls, we also consider constructor calls and public attribute accesses. Furthermore, edges are weighted by the pair $(nbCall, fctCall)$, where

$nbCall : E \longrightarrow \mathbb{Z}$ is the number of calls between the two connected classes, and
$fctCall : E \longrightarrow M$ is the set of different elements (methods, constructors, or attributes) that are called/accessed in the target class.

The DCG is obtained by merging individual graphs, each corresponding to an execution of a use case. Every case is executed one or more times with different input data. The number of executions depends on the input space of the use case. In our experiments, we use existing test cases to explore the execution possibilities. Actually, use-case executions generate traces in the form of trees where nodes are methods, constructors, or attributes and edges are calls or accesses. Each tree is transformed into a DCG by replacing the method/constructor/attribute nodes by their corresponding class nodes. Call/access edges are aggregated into inter-class edges and weights are calculated accordingly.

In the example of Figure 2, nodes $A.l, A.r, A.r$, and $A.r$ corresponding to elements (methods, constructors, or attributes) l and r of class A, are replaced by the node A. This replacement is done for all the classes. Edges $< A.l, B.m >$ and $< A.l, B.n >$ are aggregated into the edge $< A, B >$ with $nbCall = 2$ (two calls/accesses) and $fctCall = \{m, n\}$ (to two different elements m and n). Similarly, the three edges $< B.m, A.r >$ are aggregated into an edge $< B, A >$ with $nbCall = 3$. However, in this case, $fctCall = \{r\}$ because, the three calls target the same element r.

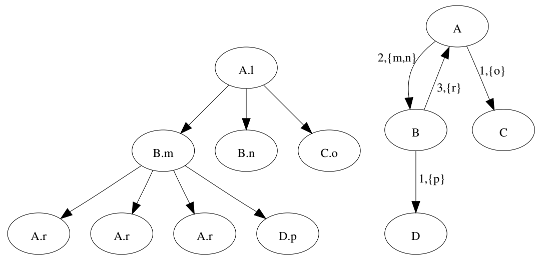

Figure 2: Example of Transforming an Execution Trace (left) into a DCG (right).

Individual DCGs of all executions are merged to produce the global DCG. Merging two DCGs $G_1(V_1, E_1)$ and $G_2(V_2, E_2)$ into a DCG $G(V, E)$ is done according to the following rules:

R_1: $V = V_1 \cup V_2$
R_2: $E = E_1 \cup E_2$
R_3: $if\ e_1 =< x, y >\in E_1, e_2 =< x', y' >\in E_2$
$x = x'$ and $y = y'$
then $for\ e =< x, y >\in E$
$nbCall_e = nbCall_{e_1} + nbCall_{e_2}$ and
$fctCall_e = fctCall_{e_1} \cup fctCall_{e_2}$

For example in Figure 3, edges $< A_3, A_1 >$ in G_1 and G_2 are merged into an edge $< A_3, A_1 >$ in G_1 with:

$nbCall_{<A3,A1>_G} =$
$nbCall_{<A3,A1>_{G_1}} + nbCall_{<A3,A1>_{G_2}} = 4.$
$fctCall_{<A3,A1>_G} =$
$fctCall_{<A3,A1>_{G_1}} \cup fctCall_{<A3,A1>_{G_2}}) = \{h, g, q\}.$

We can easily see from the three rules given above that the operator *merge* $(+)$ is commutative and associative. Therefore, merging n graphs can be done by including one graph at a time.

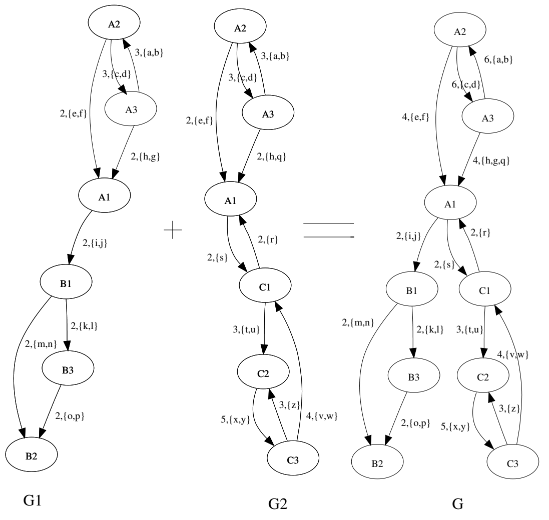

Figure 3: Example of Merging two Graphs.

4 Candidate Component Identification

In the previous section, we showed how the input of our identification process is built from the execution traces. The present section is dedicated to the three steps that allow the identification of components (see Figure 1).

4.1 Finding Candidate Groups

As stated previously, component identification is modeled as a clustering problem. For a program S with n classes, the number of all possible groups to investigate is 2^n (size of the power set $P(S)$). Some of the groups are meaningful, *i.e.,* classes in a group appear together in at least one execution trace. But, the majority are not pertinent (unrelated classes). To find only meaningful groups, we use concept lattices [4], called also Galois lattices.

Definitions : The Galois lattice [4] allows to find groups of objects sharing the same properties.

Consider two finite sets S and S', a binary relationship $R \subseteq S \times S'$ and $P(S), P(S')$ the powerset of S, respectively S'. Each element of the lattice is a pair, denoted (In, Ex), composed of a set $In \in P(S)$ and of a set $Ex \in P(S')$ verifying the two following properties:

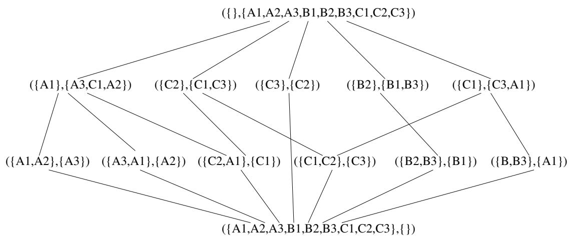

Figure 4: Lattice of the Graph of Figure 3

$Ex = f(In)$ where $f(In) = \{x' \in S' \mid \forall x \in In, xRx'\}$ is the set of all common images of the elements of Ex by relation R
$In = f'(Ex)$ where $f'(Ex) = \{x \in S \mid \forall x' \in Ex, xRx'\}$ is the set of all common antecedents of the elements of In by relation R

The Galois lattice G corresponds to the set of pairs (In, Ex).

In our case, a lattice is computed from the dynamic call graph $G(V, E)$. Both S and S' are equal to V, the set of classes of G. The binary relation R corresponds to relation E. Two classes of V are in relation E if there exists an edge in G between them.

The Galois lattice gives then the sets of classes which are in direct relation. Indeed, in a pair (In_i, Ex_i), Ex_i is the set of classes that call all the classes in In_i. The candidate component corresponds here to the set of $In_i \cup Ex_i$. For example, the lattice of Figure 4, built from the DCG of Figure 3, gives the following candidate components: $\{A1, A2, A3\}$, $\{C1, C2, C3\}$, $\{C2, C3\}$, $\{A1, A2, A3, C1\}$, $\{A1, C1, C3\}$, $\{A1, C1, C2\}$, $\{B1, A1, C1\}$, and $\{B1, B2, B3\}$.

Complexity: Let n be the size of the set S and k the upper bound of $R(s)$, such as $\forall s \in S\ |R(s)| \leq k$, in [4], Godin et al. show that the size of the lattice (the maximum number of nodes nl) is bounded by $nl \leq 2^k n$.

In our case, n corresponds to the number of classes and k to the maximum number of classes called by a class. Consequently, when dealing with large programs, the size of the lattice increases linearly with the size of the program in terms of number of classes. However, we conjecture that the maximum number of classes called by a class is not dependent on the program size. This prevents the lattice from growing exponentially and makes this step scalable.

4.2 Selecting Components

In a clustering problem that involves a set of objects V (in our case the classes of the target program), the goal is to find a partition $PA(V) = \{Co_1, Co_2, ..., Co_l\}$ of V where each Co_i is a subset of V. PA must satisfy the completeness and consistency properties, *i.e.,* any object in V must be assigned to one and only one group. Formally:

$\bigcup Co_i = V$ (completeness)
$\forall i, j \mid i \neq j, Co_i \cap Co_j = \varnothing$ (consistency)

As described above, the Galois lattice contains the set of all the meaningful groups of classes. The goal of the component selection step is to choose among the candidate groups forming the lattice, the subset that maximizes a quality function $evalComp(Co_i)$ and that satisfies the partition properties of completeness and consistency.

$evalComp(Co_i)$ of a component Co_i is calculated using the dependencies, as defined in the DCG, between classes in Co_i as well as the dependencies with other classes outside Co_i. $evalComp$ is normalized between 0 and 1, where 0 means that the classes in Co_i have no relations between them, and 1 means that all the relations of classes from Co_i are with classes from Co_i. $evalComp$ is calculated as the average of a function $eval(c_j, Co_i)$ that evaluates the contribution of a class c_j in the component Co_i. Formally:

$$evalComp(Co_i) = \sum_{c_j \in Co_i} \frac{eval(c_j, Co_i)}{|Co_i|}$$

with $eval(c_j, Co_i)$:

$$eval(c_j, Co_i) = \frac{NC(c_j, Co_i)}{NC(c_j, Co_i) + \sum_{c_l \notin Co_i} |fctCall_{<c_j, c_l>}|}$$

and $NC(c_j, Co_i)$:

$$NC(c_j, Co_i) = \sum_{c_k \in Co_i} nbCall_{<c_j, c_k>}$$

The rationale behind the function $evalComp$ is that classes inside a component must be cohesive, *i.e.,* collaborate together to perform a particular function, and their coupling with other classes must be minimal. The degree of collaboration between two classes c_j and c_k in Co_i is captured by the number of calls between them ($nbCall_{<c_j, c_k>}$). To be consistent with the composition principle, the coupling between a class c_j belonging to a component Co_i and a class c_l outside Co_i is captured by the number of different elements (services) of c_l that are called/accessed by c_j ($fctCall_{<c_j, c_l>}$).

The selection algorithm is based on $evalComp$. Let us consider the following variables and functions/procedures:

- GL: the set of candidate groups in the lattice.
- Cs: the set of selected components.
- ts: a parameter that defines the threshold for considering a candidate as having enough quality to be selected. A value of 0.5 means that classes in a component interact more often together than with other classes.
- $sort(GL)$: a procedure that sorts candidate components according to their qualities.
- $pop(GL)$: a function that returns the top-ranked candidate.
- $update(GL, c)$: a procedure that deletes the classes in a selected component from all the remaining candidates in GL.

The selection of components can be summarized as follows.

```
Algorithm: $selection(GL,\ ts)$
$sort(GL)$
$co := pop(GL)$
while $evalComp(co) > ts$ and $|GL| > 0$ do
    $Cs := Cs \cup \{co\}$
    $update(GL, co)$
    $sort(GL)$
    $co := pop(GL)$
end
return $Cs$
```

The algorithm $selection(GL,\ ts)$ consists in selecting the components that have a good cohesion and a low coupling. At each iteration the set of candidate components is sorted according to $evalComp$. The candidate with highest value is then selected. To satisfy the consistency property, all the classes of this component are removed from the remaining candidates. These are sorted

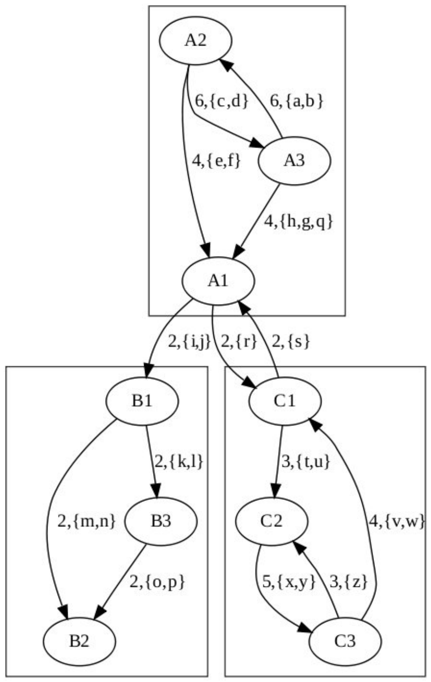

Figure 6: Indentifying component from the graph 3

again after the update operation and a new iteration is performed. The algorithm ends when one of the two following conditions is reached: all classes of the program are already in the selected components, or the quality of all remaining candidates is below a threshold, which prevents them from being considered as components. This is the case for potential glue-code classes.

Application of the selection algorithm to candidates of the lattice of Figure 4 allows to derive the three components of Figure 6. The three iterations corresponding to the selection of the three components are summarized in Figure 5.

4.3 Refining Components

Our selection algorithm can lead to a local optimum. Indeed, as we remove already selected classes from candidates, the selection of a component may impact the quality of the remaining ones. This may change the order in which they will be considered and then the global quality of the derived solution. Ideally, our algorithm should explore all possibilities and choose the best one. However as said in section 4.1, the number of candidates may be large (up to $2^k n$). Exploring all possibilities means exploring all permutations (up to $(2^k n)!$).

As an alternative, we decided to use a post-processing step, called refinement, that modifies the selected components to improve their quality. The refinement algorithm uses the function $eval$ that evaluates contribution of a class in a particular component (see Section 4.2).

To define the algorithm $refinement$, let us consider the following variables and functions/procedures:

- Cs: the set of components selected
- Cns: the set of classes not included in components after the selection step.
- $ts2$: a parameter that defines the threshold for considering a class as useful enough to be included in a component.
- $nUClasses(Cs,\ ts2)$: a function that returns all the classes having an $eval$ value, with their respective components, less than $ts2$. It also removes the returned classes from their respective components.
- $addClass(Cs,\ c,\ ts2)$: a function that adds a class c to a component in the set Cs, with which it has the best $eval$ value. If the addition degrades the global quality of the component, the class is not added. It returns a boolean to notify the success of the addition.

The principle of the refinement algorithm is as follows. First, all the classes that do not have a good contribution to their respective components ($eval$ below $ts2$) are removed from them. Then for each of these classes, we try to find a component for which they are the more useful according to $eval$ without degrading the quality of this component according to $evalComp$. We iterate on this phase until all classes are assigned to components or we reach a maximum number of iterations. It is necessary to iterate because, if initially a class c_j is not useful to or degrades the quality of a component Co_i, the introduction of another class c_k in Co_i at some moment may modify positively $eval(c_j,\ Co_i)$ and $evalComp(Co_i)$.

Iteration 1		Iteration 2		Iteration 3	
component	eval	component	eval	component	eval
$\{C1, C3, C2\}$:	0.94	$\{B2, B3, B1\}$:	0.9	$\{A2, A1, A3\}$:	0.83
$\{B2, B3, B1\}$:	0.9	$\{A2, A1, A3\}$:	0.83	$\{A1\}$:	0
$\{A2, C1, A1, A3\}$:	0.83	$\{A1, B1\}$:	0.21		
$\{A2, C1, A1, A3\}$:	0.76	$\{A1\}$:	0		
$\{C3, C2\}$:	0.61				
$\{C1, C3, A1\}$:	0.45				
$\{C1, A1, C2\}$:	0.42				
$\{C1, A1, B1\}$:	0.32				

	Selected Components	
$\{C1, C3, C2\}$	$\{C1, C3, C2\}$	$\{C1, C3, C2\}$
	$\{B2, B3, B1\}$	$\{B2, B3, B1\}$
		$\{A2, A1, A3\}$

Figure 5: Example of component selection

The refinement algorithm can be summarized as follows:

```
Algorithm: $refinement(Cs, Cns, ts2, iM)$
$iter = 0$
$Cns := nUClasses(Cs, ts2) \cup Cns$
while $Cns \neq \emptyset$ and $iter < iM$ do
    $iter := iter + 1$
    for $c \in Cns$ do
        if $addClass(Cs, c, ts2)$ then
            $Cns := Cns - \{c\}$
        end
    end
end
return $Cs$
```

5 Case Studies

In this section, we start by briefly describing our prototype. We, then, present a case study on a small-size program (39 classes). The application of our prototype to a bigger program (204 classes) is also discussed.

5.1 Prototype

The proposed approach was implemented in Java. The prototype contains three modules: (1) DCG extraction, (2) Galois lattice construction, and (3) component identification. The third module was coded from scratch by implementing selection and refinement algorithms. For the first two modules, we used/adapted frameworks.

DCG Extraction Module To capture execution traces, we used MuTT, a Multi-Threaded Tracer built on top of the Java Platform Debugger Architecture (JPDA) [8]. For a given program, MuTT generates the execution traces of each thread. We adapted MuTT to generate the DCG according to transformation and merging operations described in Section 3.

Lattice-Construction Module The Galois lattice is built using the framework Galicia [19]. Galicia proposes a set of efficient algorithms for building lattices. We generate the input for Galicia from the DCG in the form of a matrix describing the relation E between the program classes (see Section 4.1). The output can be visualized and is exported to be used by the component identification module.

5.2 Case study of Logo Interpreter

5.2.1 Target Program Description

To evaluate the feasibility of our approach, we considered an interpreter of the Logo language [18] implemented in Java. The program contains 39 classes. The program uses a graphical interface that allows to write Logo statements and a window for displaying the execution results, *i.e.,* in the form of the movements of the "turtle".

This program was used because we are familiar with the internal structure and we have a track of decisions made during development. Indeed, one of the authors of this paper was in the development team of the Logo interpreter. This knowledge allows to better evaluate the output of our component

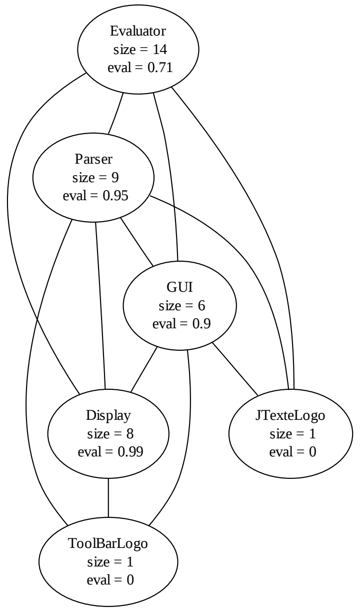

Figure 7: Component-based Architecture of the Logo Interpreter

extraction approach.

The extraction of component-based architecture took as input a DCG built from 26 execution traces corresponding to scenarios of 6 use cases. Examples of use cases are “file creation/saving”, “code writing in the editor”, “code interpretation”, etc. The execution traces cover all the 39 classes of the program. We used the following parameter values for the component identification: ts of the selection algorithm is set to 0.7; $ts2$ of the refinement algorithm is set to 0.7; maximum length of paths defining the relations between classes for building the lattice is set to 1, *i.e.,* $R = E$ (see Section 4.1); iM is set to 3.

5.2.2 Component Extraction Results.

From the performance point of view, DCG extraction was the most time-consuming step: more than 10 minutes, in part because of user interactions required when entering inputs for use cases. Construction of the Galois lattice took 0.54 seconds and produced 52 candidates. The selection and refinement steps took altogether 0.1 seconds. All these time values were collected from executions on a standard PC configuration.

From the qualitative point of view, the six following components were found in the Logo interpreter. Two of them are singletons (see Figure 7). Except for the two singletons, we name the component with the services they offer.

- Parser (9 classes): this component contains all the classes involved in the parsing such as token, tokenizer, and lexer, in addition to the class Parser. Two from the nine assigned classes should not be in this component. They are related to the component GUI.
- Evaluator (14 classes): this component is responsible for the evaluation of expression. It contains the class Evaluator and a set of classes implementing the Logo language functions except for the graphical display functions. Indeed classes implementing these functions were logically assigned to the component Display.
- Display (8 classes): this component corresponds to display functions in Logo. It contains all classes responsible for graphical display as well as the library which allows to manipulate them. One class was wrongly assigned to this component and should be in the component GUI.
- GUI (6 classes): this component corresponds to the graphical user interface of the program. As said previously, three classes of this component were wrongly assigned to the components Parser and Display.
- JTexteLogo: this component is a singleton and it manages the standard output of the program. This class is used by (and strongly connected to) the components Parser, Evaluator and GUI.
- JToolBar: this component is a singleton that manages the tool bar of the program. From a certain point of view, JToolBar should be in the component GUI. However, as it has a well defined service, it can be seen as a subcomponent of GUI or a component that offers a service to GUI.

Identified candidate components could be used to understand the behavior of Logo. Indeed, when replacing classes by corresponding components in

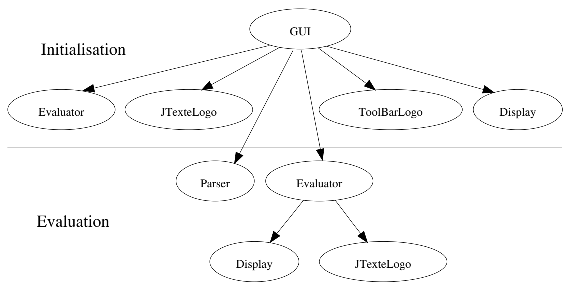

Figure 8: Execution trace

the execution traces, it is easier to observe the behavior because firstly, nodes in the traces represent functions and, secondly, traces are more compact. Each use case scenario is then associated with component interactions.

To illustrate the understanding process, let us take the example of the following use case scenario:

Actor: *Logo programmer*
Scenario:
A1: *Run the Logo interpreter*
A2: *Write the following code in the editor window*
point 1 1
write “Hello
A3: *Run the evaluation of the code from the editor window*
A4: *Close the Logo interpreter*

The execution trace corresponding to this scenario is shown in Figure 8. It contains two different phases : initialization and evaluation. In the first phase, when the Logo interpreter is run the GUI triggers the initialization of components Evaluator (more precisely, the evaluation library), JTextLogo (text standard output), ToolBarLogo (toolbar), and Display (drawing standard output) In the second phase, the typed code is parsed (Parser) and evaluated (Evaluator). During the evaluation, Evaluator calls Display to display the point at the coordinate (1,1), and JtextLogo to print the text “Hello”.

5.2.3 Conclusion

The results obtained are encouraging. Indeed, the extracted components implement independent functions. One can modify the functions of the Logo language by modifying separately these components. For example, it is possible to change the Logo syntax by modifying only the Parser component. It is also possible to replace the components (except for GUI) by others offering the same functions with a reasonable adaptation effort.

Over the 39 classes, only three were misplaced. All the misplaced classes were GUI classes that were assigned to other components. This is not surprising because of the dependencies between existing functions and the GUI, Logo being a graphical language. The case of JToolBar can be interpreted according the level of granularity in which we view the component architecture. At some level, it can be merged with GUI.

5.3 Case study of JHotDraw

5.3.1 Target Program Description

To evaluate the scalability of our approach, we applied it on the open-source framework JHotDraw. JHotDraw is a Java framework for technical and structured Graphics. The version we use contains 204 classes.

JHotDraw being a framework and not a program, the extraction of the DCG is not made directly from JHotDraw but from a demo application that uses it. The component identification took as input a DCG built from 26 execution traces corresponding to 8 use cases. The execution traces cover all the 204 classes of the program. We used the same parameter values as for the Logo Interpreter case.

5.3.2 Component Extraction Results.

As for the Logo Interpreter, the DCG extraction was done in a few minutes. Time taken for the lattice construction was longer (74.8 seconds) because of its size (833 candidate groups). Time for selection and refinement was relatively short (2.8 seconds). From the performance point of view, we can conclude that our approach is scalable.

The identification process generates 22 components containing 153 classes. The remaining 51 classes were considered as glue code. From our understanding of JHotDraw’s logic, we can claim that the majority of the 22 components are cohesive, *i.e.,* related to a particular function of JHotDraw. Examples of components are figure creation, figure saving, figure loading. However, in many cases, the components were at a fine-grained level, and could be merged to obtain more reusable components. For example, the component figure manipulation is split in 3 others components. An explanation of this phenomenon can be the fact that JHotDraw is well-designed, and in average, class coupling is relatively low.

6 Related Work

The work proposed in this paper crosscuts three research areas: architecture recovery/remodularization, legacy software re-engineering, and feature location.

Different approaches have been proposed to recover architectures from an object-oriented program. The Bunch algorithm [12] extracts the high-level architecture by clustering modules (files in C or class in C++ or Java) into sub-systems based on module dependencies. The clustering is done using heuristic-search algorithms. In [11], Medvidovic and Jakobac proposed the Focus approach whose goal is to extract logical architectures by observing the evolution of the considered systems. The approach identifies what the authors call processing and data components from reverse engineered class diagrams. Closer to our work, the ROMANTIC approach [2] extracts component-based architectures using a variant of the simulated annealing algorithm. In addition to dependencies between classes, other information sources are considered to help identifying functional components rather than logical sub-systems. Such sources include documentation and quality requirements. In [21], Washizaki and Fukazawa concentrate on the extraction of components by refactoring Java programs. This extraction is based on the class relation graphs. In the above-mentioned work, the component extraction process uses dependencies between classes/modules that are extracted using static analysis. Dependencies are not related to particular functions of the considered system which makes it difficult to relate identified components to specific functions.

Much work has already used formal concept analysis (FCA) for legacy system re-engineering. For example, FCA was used to identify objects in procedural code [16, 15, 20]. A concept lattice is built starting from the relationships between routines and data, and groups defined by the lattice are packaged into classes. In a similar problem, Siff and Reps used FCA to derive high-level modules from c files [17]. Here again, object/module identification uses static analysis, because the identified abstractions are not functional by nature.

Feature location is probably the problem that is closest to the one addressed in this paper. Many research contributions proposed solutions that are based on dynamic analysis [5] or combinations of static and dynamic analyses [3, 8, 13]. In general, static analysis uses call graphs and/or keyword querying. From the other hand, dynamic analysis consists in executing scenarios associated with features, collecting the correspondent computational units (PU), generally methods or classes, and deciding which PU belongs to which feature. The decision can be made by metrics [5], FCA [3], or a probabilistic scoring [13]. Finally, sometimes, static analysis is used to enrich the dynamic analysis. Both analyses can also be performed independently and their results combined using a voting/weighting function. The combination of static and dynamic analyses is also used in a semi-automatic process where visualizations are proposed to experts to make their decisions [1]. Like in our case, the feature location approaches use dynamic analysis and try to associate program units to scenarios. The difference, however, is that the problem of locating feature and identifying components are different in objectives and nature. In the first case, the goal is to determine code blocks, methods, or classes that are involved in a particular feature represented by a set of scenarios. For component identification a scenario may involve many features (data acquisition, processing, data store, and results displaying). The association between feature and scenario is not a one-to-one relation. Moreover, the execution of a feature may necessitate the execution of many other features, which makes it difficult to draw the boundaries. For this reason, we view the component execution as sequences of interactions between classes in an integrated dynamic call graph. Moreover, we work at the class level only, because our long-term objective is to easily package the identified components into reusable units.

7 Conclusion

In this paper we present an approach for the identification of candidate components from OO programs. These candidates can be a basis for a component-based architecture and/or can be packaged to be actually reused. Unlike other existing approaches where dependencies between classes are derived by static analysis, we use, for our component identification approach, call graphs obtained by dynamic analysis. To this end, the target program is executed on data derived from use cases, and execution traces are converted into a dynamic call graph. This guarantees that only actual functional dependencies are considered in the identification process.

Component identification is done in three steps: (1) candidates (class groups) identification using clustering by means of a Galois lattice, (2) component selection, and (3) component refinement. To deal with the algorithmic complexity of the problem, both steps (2) and (3) are based on heuristics.

We studied the feasibility of our approach on two cases. The first case is a relatively small program. Its size (39 classes) allows to deeply analyze the results. The results obtained are very satisfactory from both the performance and the qualitative points of view. The second case is a larger program that allows us to assess the scalability. Here again the results obtained were satisfactory although the components obtained were at a fine-grained level.

The studies we conducted allow us to better understand and characterize the problem of component identification. We are currently working on two variants of our approach. The first one uses a heuristic-search algorithm to explore the space of possible partitions of the set of classes. This variant uses an objective function based on $evalComp$ function. The second variant consists in searching for call patterns in the execution traces. This approach requires more executions to be efficient.

Acknowledgements

This work has been partially funded by the MDEIE Quebec, grant PSR-SIIRI107.

About the Author

Simon Allier is a PhD student at the Department of Computer Science and Operations Research of the University of Montreal and the University of South Brittany. He received a Master degree from the Univ. of Montpellier, France in 2008. His research interests are in component-based architecture recovery and component selection. His email address is alliersi@iro.umontreal.ca

Houari A. Sahraoui is Professor of Software Engineering in the Department of Computer Science and Operations Research of the University of Montreal. He obtained his Ph.D. degree in 1995 in computer science, with specialization in meta-modeling and model transformation, from the Pierre & Marie Curie University, Paris. His research interests include the application of artificial-intelligence techniques to software engineering, software visualization, object-oriented measurement and quality, and re-engineering. He has been on the program, steering, or organization committees of many international, IEEE and ACM conferences, and is member of the editorial board of three journals. He was the general chair of the IEEE Automated Software Engineering conference in 2003. His email address is sahraouh@iro.umontreal.ca

Both Simon Allier and Houari Sahraoui can be reached at DIRO, Univ de Montreal CP 6128 succ Centre-Ville, Montral QC, H3C 3J7, Canada.

Salah Sadou is an associate professor in Computer Science at Vannes University Institute of Technology. He obtained his Ph.D. degree in 1992 in computer science from the Ecole Centrale de Lyon, France. He leads the Software Evolution Group of VALORIA Lab. His general research interests are object-oriented programming, paradigms for distributed objects programming, software evolution, software architectures, model-driven engineering and component-based software engineering. His email address is salah.sadou@univ-ubs.fr. He can be reached at VALORIA Lab., Yves Coppens Research Center, University of South Brittany, B.P 573, 56017 Vannes CEDEX France.

References

[1] Johannes Bohnet and Jürgen Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *SOFTVIS*, pages 95–104, 2006.

[2] Sylvain Chardigny, Abdelhak Seriai, Dalila Tamzalit, and Mourad Oussalah. Quality-driven extraction of a component-based architecture from an object-oriented system. In *CSMR*, pages 269–273, 2008.

[3] Thomas Eisenbarth, Rainer Koschke, and Daniel Simon. Locating features in source code. *IEEE Trans. Software Eng.*, 29(3):210–224, 2003.

[4] Robert Godin, Guy W. Mineau, Rokia Missaoui, Marc St-Germain, and Najib Faraj. Applying concept formation methods to software reuse. *International Journal of Software Engineering and Knowledge Engineering*, 5(1):119–142, 1995.

[5] Orla Greevy and Stéphane Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *9th European Conference on Software Maintenance and Reengineering (CSMR 2005), 21-23 March 2005, Manchester, UK, Proceedings*, pages 314–323. IEEE Computer Society, 2005.

[6] Réda Kadri, Franois Merciol, and Salah Sadou. CBSE in Small and Medium-Sized Enterprise: Experience Report. In *CBSE*, 2006.

[7] P. Kruchten. The 4+1 view model of architecture. *IEEE Software*, 12(6):42–50, 1995.

[8] Dapeng Liu, Andrian Marcus, Denys Poshyvanyk, and Vaclav Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In R. E. Kurt Stirewalt, Alexander Egyed, and Bernd Fischer, editors, *ASE*, pages 234–243, 2007.

[9] D.H. Lorenz and J. Vlissides. Designing components versus objects: a transformational approach. In *ICSE*, May 2001.

[10] Mark W. Maier, David Emery, and Rich Hilliard. Ansi-ieee 1471 and systems engineering. *System Engineering*, 7(3):257–270, 2004.

[11] Nenad Medvidovic and Vladimir Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engineering*, 13(2):225–256, 2006.

[12] Brian S. Mitchell and Spiros Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Comput.*, 12(1):77–93, 2008.

[13] Denys Poshyvanyk, Yann-Gal Guhneuc, Andrian Marcus, Giuliano Antoniol, and Vclav Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. volume 33, pages 420–432, 2007.

[14] Roger S. Pressman. *Software Engineering: A Practitioner's Approach.* McGraw-Hill, 6th edition, 2004.

[15] Houari A. Sahraoui, Hakim Lounis, Walcélio Melo, and Hafedh Mili. A concept formation based approach to object identification in procedural code. *Automated Software Engineering*, 6(4):387–410, 1999.

[16] Houari A. Sahraoui, Walcélio Melo, Hakim Lounis, and Francois Dumont. Applying concept formation methods to object identification in procedural code. In *ASE*, pages 210–218, 1997.

[17] Michael Siff and Thomas W. Reps. Identifying modules via concept analysis. *IEEE Transaction on Software Engineering*, 25(6):749–768, 1999.

[18] Allier Simon, Grollemund Natan, Lalluque Renaud, and Delfour Kevin. http://naitan.free.fr/logo/.

[19] Petko Valtchev, David Grosser, Cyril Roume, and Mohamed Rouane Hacene. Galicia: an open platform for lattices. In *ICCS*, pages 241–254. Shaker Verlag, 2003.

[20] Arie van Deursen and Tobias Kuipers. Identifying objects using cluster and concept analysis. In *ICSE*, pages 246–255, Los Alamitos, CA, USA, 1999.

[21] Hironori Washizaki and Yoshiaki Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Sci. Comput. Program.*, 56(1-2):99–116, 2005.