Restructuring Object-Oriented Applications into Component-Oriented Applications by Using Consistency with Execution Traces

Simon Allier, Houari Sahraoui, Salah Sadou, Stéphane Vaucher

Restructuring Object-Oriented Applications into Component-Oriented Applications by using Consistency with Execution Traces

Simon Allier[1,2], Houari A. Sahraoui[1], Salah Sadou[2], and Stéphane Vaucher[1]

[1] DIRO, Université de Montréal, Canada
[2] VALORIA, South-Brittany University, Vannes, France
{alliersi,sahraouh,vauchers}@iro.umontreal.ca,sadou@univ-ubs.fr

Abstract. Software systems should evolve in order to respond to changing client requirements and their evolving environments. But unfortunately, the evolution of legacy applications generates an exorbitant cost. In this paper, we propose an approach to restructure legacy object-oriented applications into component-based applications. Our approach is based on dynamic dependencies between classes to identify potential components. In this way, the composition is dictated by the context of the application to improve its evolvability. We validate our approach through the study of three legacy Java applications.

1 Introduction

An intrinsic characteristic of software, addressing a real world activity, is the need to evolve in order to satisfy new requirements. Resulting from empirical studies, Lehman's first law states that software should evolve else it becomes, progressively, less satisfactory [10]. Although old, this law has never been contradicted. The required reactivity (increasingly growing) of software applications, supports for business processes which are evolving more and more quickly, has even increased the scope of this law as the years go by. Maintenance is now, more than ever, an inescapable activity, the cost of which is ever increasing. Estimated at approximately 50 % to 60 % of the software total cost in the eighties and nineties [11, 14]; recent studies now evaluate this cost at being between 80 % and 90 % [4, 18]. This high cost has undoubtedly been an effective catalyst for the emergence of new programming paradigms. Modular languages, then object-oriented languages, and more recently component-oriented programming, have always had as first justification, the significant increase in maintainability level. These new approaches can be used to build new applications. But what about legacy applications? In this case one can use the techniques of reverse engineering to transform the structure of the application, without changing its functionality, so that it conforms to the new paradigm.

In the past we presented a work that allows a company to organize its source code into reusable components [7]. In this work, identifying parts to put together in order to make a component was left in charge of engineers. What we

propose is to automatically identify, in the case of object-oriented applications, classes to be grouped to form a component. Some approaches for identifying components [20] or high-level architectures [16, 13] already exist. These works generally use the dependencies, between program elements, extracted by static analysis to identify highly cohesive and weakly coupled object-like structures. For components, the idea is to group together classes that contribute to the same functions. Generally, the restructuration of an application aims at improving its maintainability, including its evolvability. And often evolutions have a functional scope. But the classes used for building an application have in most cases a bigger scope than what is needed by the latter. So, unlike the approach based on static dependencies, we promote the dynamic dependencies for the aim of maintainability improvement. Thus, we clam that the most reliable way to determine which class contributes to which function is to execute the program with individual functions. Traces corresponding to execution scenarios could be analyzed to extract functional dependencies.

The rest of the paper is organized as follows: Section 2 describes the steps that constitute our approach. Scenarios execution and trace extraction are explained in Section 3. Section 4 gives details of our approach for identifying groups of classes that represent potential components. We provide three case studies in Section 5. Before concluding in Section 7, we describe some related works is in Section 6.

2 Approach Overview

We view a component as a group of classes collaborating to provide a system function. The interfaces provided and required by a component are the method calls respectively from and to classes belonging to other components.

So, we propose an approach for identifying components using traces obtained by executing scenarios corresponding to system use cases. The identification of candidate components consists of clustering the classes of the target system in such a way that classes in a group appear frequently together in the execution traces. In the same time, we also try to minimize the coupling between components.

Thus, the identification becomes an optimization problem where the goal is to identify groups of classes whose interactions are the most consistent with the execution traces. As this is a NP-hard graph partitioning problem, we use heuristic search to find a near-optimal solution.

Therefore, our approach is structured in three steps (see Figure 1):

1. Starting from a set of use case scenarios generate the execution traces. This step allows identifying dependencies between classes;
2. Produce a preliminary set of candidates using a global search. At this stage we use a genetic algorithm to produce this initial solution;
3. Refine the component candidates using a local search. We use a simulated annealing algorithm in order to achieve the local search.

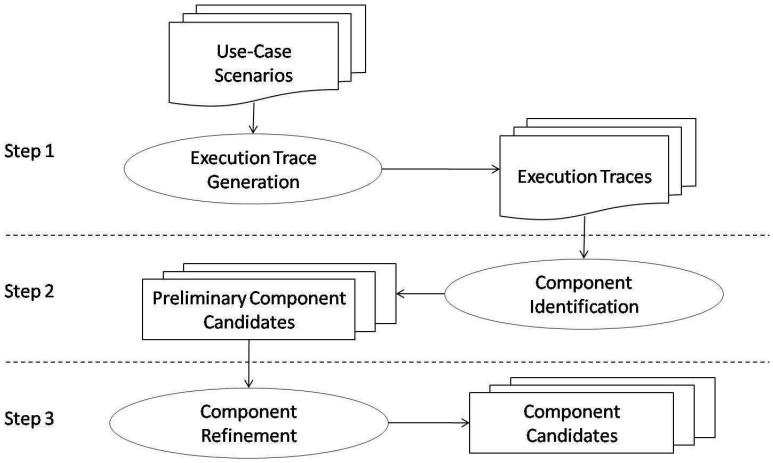

Fig. 1. A three-step process for component identification

At the end of the identification process, groups that have a consistency with the traces below a predefined threshold are candidates to be packaged as components.

The execution traces are considered as the reference that guides the search for an identification solution. In the first step of the search, a population of potential groupings is created. Following an evolutionary algorithm, new groupings are derived that better match the interactions contained in the execution traces. This step serves primarily to find the region of the search space that has the highest potential to contain the best solution. Another step, in the form of a local search, allows to explore this region to find the near-optimal solution.

This approach does not aim to identify reusable components. Indeed, component identification, in our case, is guided by the functional logic of the considered application. Restructuring the application using the identified components has the sole purpose of improving its maintainability. Thus, the goal of our identification process is not to fully re-architect a system into components. Our objective is rather to find groups of classes that could be packaged for reuse purpose.

It is true that in the case of a company, who work in a particular application domain, identified components can be considered reusable. But in this case, the execution traces must be obtained from several of its applications.

The extraction of the execution traces is described in Section 3. The two steps of the component identification are detailed in Section 4.

3 Execution Trace Generation

For the purpose of component identification, we are interested in finding dynamic relationships between classes. Hence, an execution trace (or call tree) is for us a directed tree $T(V, E)$ where V is a set of nodes and E, a set of edges between

nodes. Each node V_i represents a class of the system (Cl_i). An edge $\langle V_i, V_j \rangle$ indicates that an object of class Cl_i calls a method of an object of type Cl_j. The root of $T(V, E)$ corresponds to the entry point of the system or a thread run by the system.

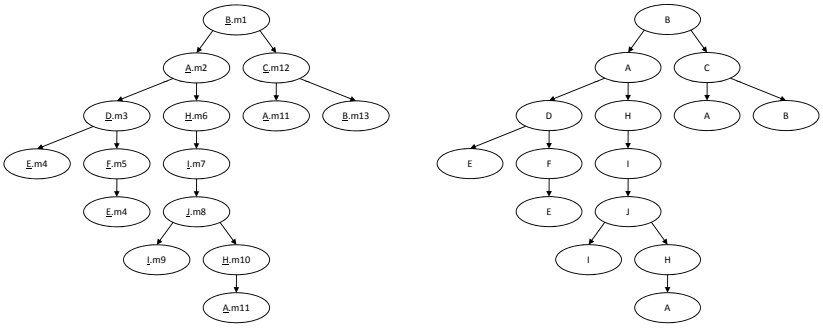

Fig. 2. Example of Execution Trace. (Left) Method-Call Tree. (Right) Corresponding Dynamic Relationships between Classes

The execution traces are obtained by capturing the calls between instance of classes during an execution of a use case scenario. Every thread, created during the execution, produces an execution trace. In this initial version of the traces, nodes are labeled by both the actual types of the objects that are called and the methods called as showed in Figure 2(left). For example, when execution method m1, object B, instance of class B, called method m2 with as receiver object A, instance of class A. A, in turn, called m3, with as received D of class D.

In a second phase, nodes of the execution traces are relabeled by the classes corresponding to the called objects. The relabeling process is straightforward. Indeed as the call tree is generated dynamically, the concrete type of each object is recorded. This second phase produces traces where nodes are classes, and edges, dynamic relationships between classes. In the example illustrated in Figure 2 (right), the sequence <B.m1, A.m2, D.m3,> is replaced by <B, A, D>.

The identification of components from execution trace is relevant only if the execution traces cover all the functions of the system. Therefore, to extract the traces, we systematically apply all the recorded execution scenarios in the documentation.

4 Component Identification

As mentioned previously, the identification of components is modeled as a clustering problem. Indeed, the goal is to find the best partitioning $P_i(V) = \{C_1, ..., C_n\}$ where V is the set of classes in the system and a C_j is a a com-

ponent candidate containing a subset of V. $P_i(V)$ needs to satisfy the following completeness and consistency properties:

- $\bigcup_{0<j\leq n} C_j = V$ (completeness)
- $\forall j, k | j \neq k, C_j \cap C_k = \{\emptyset\}$ (consistency)

Exploring every possible clustering configuration cannot be done efficiently as we would need to consider every possible combination (NP-hard). We therefore propose using meta-heuristic search algorithms to find a near-optimal solutions. These algorithms are generic and are applied to a specific domain (in our case, graph partitioning) by defining the possible movements in the search space (transformations) and a fitness function that will be maximized.

In our approach, the search for a component-based architecture is implemented using a hybrid search [8] which combines two different meta-heuristics: genetic algorithm (GA) and simulated annealing (SA). GA is a global search heuristic that applies changes to multiple solutions (called populations) and returns a solution that is globally near-optimal. This solution is then used as the initial solution of the SA algorithm, a local search algorithm. SA is called a local search algorithm because it explores the neighbourhood of a solution. Its output is our final solution. Both algorithms use the same solution representation and fitness function.

4.1 Solution and Space Representation

A solution for both algorithms can be any set of candidate components which respects the criteria of completeness and consistency. These candidates are represented by the set of classes they contain. In the example presented in Figure 2, the execution trace corresponds to a system with a set of classes $A, B, C, D, E, F, H, I, J$. One possible solution maybe: $\{\{A, C, D\}, \{E, J\}, \{H, I, B, F\}\}$. But there are many alternatives. In the following subsections we'll see how to choose the best alternatives.

4.2 Fitness Function

The fitness function used (Equation 1) evaluates the quality of a partition considering both the internal cohesion of components and the level of inter-component coupling of every component C (C can also be seen as a set consisting of classes that it gathers). The function takes as input A (for architecture), the set of component candidates proposed by the solution, and calculates the weighed average of the fitness of individual components (Cl being the set of classes in the system). The fitness of individual components depends mostly on their cohesion (Equation 2) unless the coupling level is too high in which case the fitness score is heavily penalized. The threshold cm used corresponds to the average coupling of all the classes in the system.

$$evalArch(A) = \frac{1}{|Cl|} \sum_{C \in A} (evalComp(C) * |C|) \quad (1)$$

$$evalComp(C) = \begin{cases} evalCoh(C)/2 & if\ evalCoupling(C) < cm \\ evalCoh(C)/2 + 0.5 & otherwise. \end{cases} \quad (2)$$

Cohesion A good component should include classes that interact with one another to provide a specific set of functionalities; the strength of these interactions are what we call cohesion. The internal cohesion measure (Equation 3) evaluates how close are the different classes in the execution traces. It calculates the average distance between pairs of classes belonging to this component in all the traces. The distance between two classes a and b (Equation 4) is the average distance between instances of $obj(a)$ of a and $obj(b)$ of b.

To reduce the complexity of exploring the traces when calculating distances (number of edges), we use a constant d which indicates the maximum interesting distance between two classes. In other words, if we consider that the distance in the call graph, between two classes, that is acceptable to consider is for example $d = 5$, then we normalize the actual distance n using d (dividing it by 5). For distances less than 5, it will weigh less than 1 (acceptable) and for 5 and more is given a weight of 1 (worst penalty). The distance $distMin(C, x, b)$ is the minimal distance between a node corresponding to an instance x of the class a and any instance of the class b in the execution trace where x appears. If no path with a length less than d is found, the distance is considered as d. all the node of the path are an instance of a class of C and such as the leng of the path is lower to d, otherwise $distMin(C, x, c)$ return d.

$$evalCoh(C, d) = \begin{cases} \frac{1}{|C|^2 - |C|} \sum_{x \in C} \sum_{y \in C, y \neq x} dist(x, y, C) & if \quad |C| > 1 \\ 1 & if \quad |C| = 1 \end{cases} \quad (3)$$

$$dist(a, b, C) = \frac{1}{d|obj(a)|} \sum_{x \in obj(a)} distMin(C, x, b) \quad (4)$$

The number of classes into the same component increases with d. The choice of the best value of d can be done with small examples of applications from the domain and some experience in the domain.

Coupling One of the strengths of component-based development is that its components are loosely coupled and can be mixed and matched to build systems. The function $evalCoupling(C)$ (Equation 5 evaluates the level of coupling between components by counting the number of classes that are *connected* to a component (either calling or called). Classes that are part of the component are ignored.

$$evalCoupling(C) = |\bigcup_{x \in C} connected(x)| \quad (5)$$

For the example presented in Figure 3, the solution $S = \{\{A, B, C\}, \{D, F\}, \{E, H, I, J\}\}$ would produce the following metrics with $d = 3$:
$cm = 2.66$
$evalCoh(\{A, B, C\}) = 0.47$
$evalCoh(\{D, F\}) = 0.33$
$evalCoh(\{E, H, I, J\}) = 0.7$
$evalCoupling(\{A, B, C\}) = 2$
$evalCoupling(\{D, F\}) = 2$
$evalCoupling(\{E, H, I, J\}) = 4$
$evalArch(S) = \frac{((0.47/2)*3+(0.33/2)*2+((0.7/2+0.5)*4))}{9} = 0.49$

In the following, we'll see how to determine the possible compositions.

4.3 Global Search

A genetic algorithm is a global meta-heuristics that emulates the concept of evolution. In order to find a solution from a population, it starts with a population (P_0) containing a set of solutions (called chromosomes) and simulates the passing of generations on this population. This initial population is a randomly generated. For every iteration of the algorithm, a new population (P_{i+1}) is produced by *selecting* pairs of chromosomes ((c_1, c_2)) from P_i and applying a crossover and/or a mutation transformation to these pairs with a certain probability. For this work, we systematically add the best chromosome of a generation to the next generation. The precise algorithm (Algorithm 1) uses three inputs: $MaxIter$, $MaxNiter$ and $MaxSize$. $MaxIter$ is the maximum number of generation for the evolution. $MaxNiter$ defines maximum number of generations where no improvement is accepted. Finally, $MaxSize$ defines the maximum size for a population.

Selection. The probability of selecting a chromosome c from the current population P depends on its quality with regards to the other chromosomes. This probability is given by the function:

$$Ps(c, P) = \frac{evalArch(c)}{\sum_{a \in P} evalArch(a)}$$

This way of selecting components, called elitism, allows to give more chance to the fittest components to be selected.

Crossover. The "classic" crossover transformation consists of spliting two chromosomes c_1 and c_2 into two parts and merge the first part of c_1 (respectively the second part of c_1) with the second part of c_2 (respectively the first part of c_2). However, this crossover might generate a solution that does not respect the constraints of completeness and consistency. Indeed, some classes could exist in more than one component in the new generated chromosomes or do not exist at all. To preserve the two above-mentioned properties, we propose the following variation:

Algorithm: $genetic(MaxIter, MaxNiter, MaxSize)$
let $iter := 0; niter := 0$
create a initial population P_0
let $Best := min_{c \in P} evalArch(c)$
while $(iter < MaxIter)$ *and* $(niter < MaxNiter)$ **do**
  eval P_{iter}
  let $P_{iter+1} := \{\emptyset\}$
  while $P_{iter+1} < MaxSize$ **do**
    Select $c_1, c_2 \in P_{iter}$
    Crossover c_1, c_2 with probability p_c to c'_1, c'_2
    Mutate c'_1, c'_2 with probability p_m to c''_1, c''_2
    $P_{iter+1} := P_{iter+1} \cup \{c''_1, c''_2\}$
  end
  let $BestLocal = min_{c \in P_{iter}} evalArch(c)$
  $P_{iter+1} := P_{iter+1} \cup \{BestLocal\}$
  if $evalArch(BestLocal) < evalArch(Best)$ **then**
    $Best := BestLocal$
    $niter := 0$
  end
  $iter++; niter++$
end
return $Best$

Algorithm 1: Genetic algorithm

- Divide the chromosome c_1 (respectively c_2) into two parts c_{11} and c_{12} (respectively c_{21} and c_{22}), each containing a subset of components.
- Create a chromosome c'_1 by insert c_{11} between c_{21} and c_{22} (respectively c'_2 by inserting c_{21} between c_{11} and c_{12})).
- Delete in c_{21} and c_{22} all the classes appearing in c_{11} (respectively in c_{11} and c_{12} all the classes appearing in c_{22}).

For example, the chromosomes:
$c_1 = \{\{A, C, I\}, \{E, J\}, \{D, H, B, F\}\}$
$c_2 = \{\{A, H\}, \{B, C, D, E\}, \{F\}, \{I, J\}\}$
partitioned into:
$\{\{\mathbf{A,C,I}\}\}$ and $\{\{E, J\}, \{D, H, B, F\}\}$ for c_1,
$\{\{\mathbf{A,H}\}, \{\mathbf{B,C,D,E}\}$ and $\{F\}, \{I, J\}\}$ for c_2.
produces the two chromosomes:
$c'_1 = \{\{H\}, \{B, D, E\}, \{\mathbf{A,C,I}\}, \{F\}, \{J\}\}$, and
$c'_2 = \{\{I\}, \{\mathbf{A,H}\}, \{\mathbf{B,C,D,E}\}\{J\}, \{F\}\}$.

Mutation. There are three type of mutation applicable to a chromosome:

- Split of a component in two components;
- Merge of two components;
- Move of a class from a component to another.

The type of the mutation is selected randomly as they are components involved in the mutation.

All three mutations produce solutions that preserve the properties of completeness and consistency. For example, $c = \{\{A, C, D\}, \{E, J\}, \{B, F, H, I\}\}$ could be mutated in:
$c_{merge} = \{\{A, C, D\}, \{B, E, F, H, I, J\}\}$ or into
$c_{move} = \{\{A, C\}, \{D, E, J\}, \{B, F, H, I\}\}$.

4.4 Local Search

GA can explore different solutions in a large search space to produces a solution that is globally near-optimal. This solution is then used by SA as a starting point for a fine grained exploration of its neighbourhood with the objective of refining it. The algorithm is presented Algorithm in 2. SA manipulate only one solution (s) at a time. At each iteration of the algorithm, this solution is compared to a neighbour (s_{neigh}) generated by a function $Neigh(x)$. When s_{neigh} is better than s as measured by the fitness function ($evalArch$), it replaces it. Otherwise, it can be accepted with a small probability which decreases as the algorithm progresses. This element of randomness is included to avoid falling into a local optimum.

Neighbour Function The neighbourhood function ($Neigh(s)$) uses the mutation of the genetic algorithm to produce a neighbour.

Algorithm: $SimulatedAnnealing(s, Tp, delta, tMin, iter, cof)$
let $Best := s$
while $Tp > tMin$ **do**
  for $i = 0; i < iter; i++$ **do**
    let $s_{neigh} := Neigh(s)$
    let $delta := evalArch(s) - evalArch(s_{neigh})$
    if $(delta < 0)$ *or* $(random < e^{\frac{-delta}{Tp}})$ **then**
      $s := s_{neigh}$
    end
    if $evalArch(s_{neigh}) < evalArch(Best)$ **then**
      $Best := s_{neigh}$
    end
    $Tp := cof * Tp$
  end
end
return $Best$

Algorithm 2: Simulated Annealing algorithm

5 Case Study

In this section, we present and discuss the results obtained on three systems of different size (respectively 40, 73 and 221 classes).

5.1 System Descriptions

Our approach was evaluate on three systems. The first is an interpreter of the language Logo [3]. It has a graphical interface which allows writing the code and a window which shows the result graphically. This programs contains 40 classes. Jeval [4] is the second program. It is an expression interpreter. It contains 73 classes. Finally the last system is Lucene [5], a high-performance, full-featured text search engine. Lucene contains 221 classes.

	number of executions	trace
Logo	8	19
Jeval	9	9
Lucene	19	59

Table 1. Capture of the executions traces

5.2 Extraction of Traces

The extraction of execution traces was implemented using MuTT [12]. MuTT is a Multi-Threaded Tracer built on the top of the Java Platform Debbuger Architecture. For a given program execution, MuTT generates an execution trace for each thread.

For the three system, the extraction of the execution traces were generated as follows:

- Logo: The execution traces were obtained by executing different scenarios of use case. The definition of use cases and execution scenarios was easy, because, one of the authors of this paper was in the development team of the Logo interpreter.
- Jeval, Lucene: The execution traces of Jeval and Lucene were derived by executing different scenarios of use cases. Scenarios was obtained by the test cases defined for these systems. We ensured that the test cases cover well the use-case scenarios.

Table 1 gives for every system the number of execution scenarios and the number of generated execution traces. There are more traces than scenarios because a trace is generated for each thread.

[3] http://naitan.free.fr/logo/
[4] http://jeval.sourceforge.net
[5] http://lucene.apache.org/

5.3 Result

The identification results for the three systems are presented in Table 2. For each system, it shows the number of identified components and the numbers of those who are related to the application and those who are not. A component is *related* if it contains the classes that provide a system function. It is considered as *not related* otherwise.

	number of component	related	not related
Logo	5	3	2
Jeval	5	5	0
Lucene	25	16	9

Table 2. Results of component identification

Logo Interpreter This system produces 5 components. Component *Library* implements the basic functionality of the language Logo (Math, String, ...). *Display* is composed of the classes responsible for the display of the instructions of the language Logo. Both components implement only one function. The third component *Evaluator GUI* provided two interrelated functions: the evaluation and the GUI for the result of the evaluation. The two other components do not contain functionally related classes. They both contain classes related to error management and other classes that plays the role of glue code between the three other components.

Jeval This system is partitioned into 5 components. Of these 5 components two represent respectively the library of the mathematical functions (sin, log, ...) and the library of the string functions. All the contained classes are related to the functions. The three other components contain classes necessary for respectively the parsing, the interpretation, and the mathematical operators evaluation. Depending on the viewpoint, these components could be merged.

Lucene From the 25 components, 16 are good and 9 bad. Four of the good components contain clear single functions. For example, *QueryParser* contain only classes responsible for the parsing of the search queries. The others 12 good components provide only one function, but with few classes missing. Here again, some could be merged if the goal is to obtained coarse-grained components. For example, the indexation function is split into subfunctions (5 components). Finally 9 identified components have no clear function.

As mentioned in section 2, the goal of our identification process is to find group of classes that could be packaged for reuse purpose. It is then normal that some of identified components are not considered as good. When putting a threshold on *EvalComp*, almost all the bad components will not be considered.

5.4 Components as a Behavioral Understanding Aid

Identified candidate components could be used to understand the behavior of a system. In the case of Logo Interpreter, when classes are grouped by their corresponding components in the execution traces, one can understand the behavior of the system. Indeed, the obtained nodes in the traces represent the system functions and, the links represent the function interrelations. Each execution scenario is then associated with a component interaction scenario.

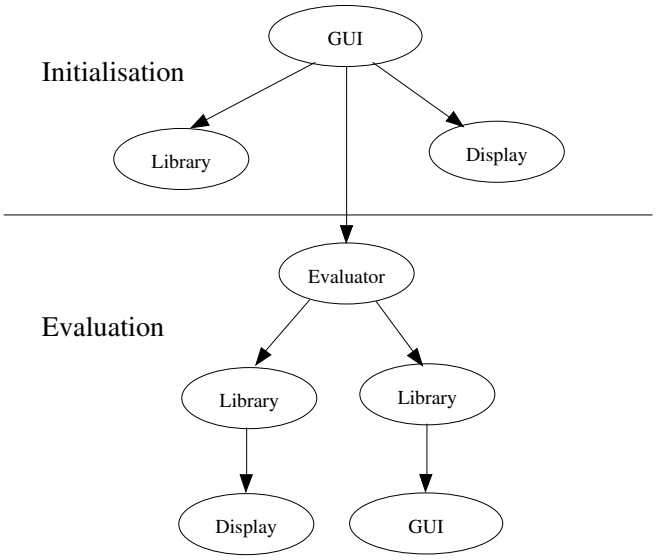

Fig. 3. Execution trace

To illustrate the behavior 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 3. In this trace the component *EvaluatorGUI* was manually divided into two components *Evaluator* and *GUI*. This trace contains two different phases : initial-

ization and evaluation. In the first phase, when the Logo interpreter is run the GUI triggers the initialization of components Evaluator Library and Display (drawing standard output). In the second phase, the typed code is parsed and evaluated (Evaluator). During the evaluation, Evaluator calls twice Library: the first time for the function point and second time for the function write. The function point call the component Display to display the point at the coordinate (1,1), and the function call the component GUI to print in the standard output the text "Hello".

5.5 Discussion

The results of this case study are satisfactory. Indeed even if components identified are not all in the "related" category, the majority provide a unique feature and by spliting or merging the others, it is easy to obtain "related" component. Furthermore, this case study revealed a possible limitation of our approach. Indeed, our approach is designed to treat all the classes of the system as potential parts of components. It does not consider explicitly the case of glue-code classes. Detecting such classes and excluding them from the partitioning will certainly improve the identification results.

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 [16] extracts the high-level architecture by clustering modules (files in C or class in C++ or Java) into subsystems based on module dependencies. The clustering is done using heuristic-search algorithms. In [15], 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 [20], 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. Thus, the components identified have a general scope and are not dedicated to the application. Whereas in the case of our approach, identification is guided by the context of the application, which will facilitate its maintenance.

Our approach use heuristic-search methods [8], genetic algorithm and simulated annealing. Search-based methods is widely applied to solve problems similar to ours. For example, Seng [19] improves the design of an 00 code with a fitness function that quantify the improvements in design metrics. To this end, a genetic algorithm is used. In [9], Kessentini uses meta-heuristiques to transform models by examples. More close to our work, [2] use a variant of the simulated annealing for extract component-based architectures.

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 [6] or combinations of static and dynamic analyses [3, 12, 17]. 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 [6], FCA [3], or a probabilistic scoring [17]. 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 features 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.

7 Conclusion

Our main objective in restructuring an object-oriented application into a component-oriented application is the improvement of its evolvability. Thus, unlike other existing approaches where dependencies between classes are derived by static analysis, we used, for our component identification approach, method call trees obtained by executing use case scenarios on the application. This guarantees that only functional dependencies are considered in the component identification. Indeed, application's evolutions have most often a functional scope. Moreover, the execution traces, obtained thanks to the use cases, limit the analysis of dependency only to the space covered by the application. While the classes, which are often generic, cover a wider space.

In the past we presented a work that allows a company to organize its source code into reusable components [7]. This consisted in a reorganization of the development life-cycle and in the use of the UML2 component model in order to wrap a code corresponding to a component. Although in the case of our experimentation with our industrial partner, the engineers know very well the existing code, identification remains a tedious job. Moreover, in several cases of incorrect component identification, the cause was a reflex related to their long experience with the object-oriented approach.

The work presented here is complementary to that presented above. It is an aid in identifying components by grouping classes. As our approach relies on execution traces of an application, the proposed grouping is necessarily adapted to this application. What goes in the direction of improving the maintainability of the application. In the case of our industrial partner, applications are built to the same scope (Geographical information systems) using their library of classes. To build their library of components, we should grouping classes by using traces from all their applications. After that, we plan to use another work, that we have already made, in order to automatically select components [5].

Our approach comes just before building the components. It only proposes the classes that must go together. Thus, its use is possible with any model of components.

References

1. J. Bohnet and J. Döllner. Visual exploration of function call graphs for feature location in complex software systems. In *SOFTVIS*, pages 95–104, 2006.
2. S. Chardigny, A. Seriai, D. Tamzalit, and M. Oussalah. Quality-driven extraction of a component-based aachitecture from an object-oriented system. In *CSMR*, pages 269–273, 2008.
3. T. Eisenbarth, R. Koschke, and D. Simon. Locating features in source code. *IEEE Trans. Software Eng.*, 29(3):210–224, 2003.
4. L. Erlikh. Leveraging legacy system dollars for e-business. *IEEE IT Professional*, 2(3), 2000.
5. B. George, R. Fleurquin, and S. Sadou. A methodological approach for selecting components in development and evolution process. *Electronic Notes on Theoretical Computer Science (ENTCS)*, 6 (2):111–140, January 2007.
6. O. Greevy and S. Ducasse. Correlating features and code using a compact two-sided trace analysis approach. In *CSMR*, pages 314–323. IEEE Computer Society, 2005.
7. R. Kadri, F. Merciol, and S. Sadou. CBSE in Small and Medium-Sized Enterprise: Experience Report. In *CBSE*, June 2006.
8. V. Kelner, F. Capitanescu, O. Léonard, and L. Wehenkel. A hybrid optimization technique coupling an evolutionary and a local search algorithm. *J. Comput. Appl. Math.*, 215(2):448–456, 2008.
9. M. Kessentini, H. Sahraoui, and M. Boukadoum. Model transformation as an optimization problem. In *MODELS*, pages 159–173, Berlin, Heidelberg, 2008. Springer-Verlag.
10. M. Lehman and L. Belady. Program evolution: Process of software change. *London: Academic Press.*, 1985.

11. B. P. Lientz and E. B. Swanson. Problems in application software maintenance. *Communiactions of the ACM*, 24(11), 1981.
12. D. Liu, A. Marcus, D. Poshyvanyk, and V. Rajlich. Feature location via information retrieval based filtering of a single scenario execution trace. In R. E. K. Stirewalt, A. Egyed, and B. Fischer, editors, *ASE*, pages 234–243, 2007.
13. O. Maqbool and H. Babri. Hierarchical clustering for software architecture recovery. *IEEE Trans. Softw. Eng.*, 33(11):759–780, 2007.
14. J. McKee. Maintenance as function of design. In *AFIPS National Computer Conference*, pages 187–193, 1984.
15. N. Medvidovic and V. Jakobac. Using software evolution to focus architectural recovery. *Automated Software Engg.*, 13(2):225–256, 2006.
16. B. S. Mitchell and S. Mancoridis. On the evaluation of the bunch search-based software modularization algorithm. *Soft Comput.*, 12(1):77–93, 2008.
17. D. Poshyvanyk, Y.-G. Guéhéneuc, A. Marcus, G. Antoniol, and V. Rajlich. Feature location using probabilistic ranking of methods based on execution scenarios and information retrieval. *IEEE Trans. Software Eng.*, 33(6):420–432, 2007.
18. R. C. Seacord, D. Plakosh, and G. A. Lewis. Modernizing legacy systems: Software technologies, engineering processes, and business practices. *SEI Series in Software Engineering*, 2003.
19. O. Seng, J. Stammel, and D. Burkhart. Search-based determination of refactorings for improving the class structure of object-oriented systems. In *GECCO*, pages 1909–1916, New York, NY, USA, 2006. ACM.
20. H. Washizaki and Y. Fukazawa. A technique for automatic component extraction from object-oriented programs by refactoring. *Sci. Comput. Program.*, 56(1-2):99–116, 2005.