complementary-material-gecco

2020

1 Criteria Computation

Our approach uses five criteria. They are formalized in what follows.

Coupling - To compute coupling between candidate microservices, we rely on coupling using static information. The Equation 1 presents the coupling computed for each microservice candidate (MS_c). In addition, sc is the number of calls present in the body of v_i method and particularly made to the v_j method, where $(v_i, v_j) \in E$ (edges set in the graph that represent the legacy system).

$$\delta(MS_c) = \sum^{v_i \in MS_c \wedge v_j \notin MS_c} sc(v_i, v_j) \tag{1}$$

In summary, δ function showed in Equation 1 is the number of static calls from methods within a MS_c to another microservice candidate in the same MSA (microservices architecture).

Equation 2 describes how to compute the overall coupling of all MS_c in the MSA. Basically, it is the sum of the couplings associated with all microservice candidates.

$$Coupling = \sum^{\forall MS_c \in MSA} \delta(MS_c) \tag{2}$$

Cohesion. The cohesion of a microservice candidate is computed by dividing the number of the static calls between methods within the microservice boundary (i.e., the set of methods assigned to the candidate) by all possible existing static calls. This declared way of measuring cohesion indicates how strongly related the methods are within a microservice candidate.

$$ce(v_i, v_j) = \begin{cases} 1, if sc(vi, vj) > 0 \\ 0, \ otherwise \end{cases} \tag{3}$$

In order to compute it, the ce function is defined in Equation 3 as a boolean function indicating the existence of at least a static call.

$$C(MS_c) \;=\; \frac{\sum^{\forall v_i \in MS_c \wedge v_j \in MS_c} ce(vi, vj)}{\frac{|MS_c|(|MS_c|-1)}{2}} \tag{4}$$

The cohesion of a microservice candidate is presented in Equation 4, where $|MS_k|$ is the cardinality of a MS_c. Basically, Equation 4 divides the number of static calls by the number of all possible dependencies between methods of a candidate microservice. In this sense, the denominator of Equation 4 is the combination two-by-two of all methods within a MS_c.

Lastly, Equation 5 defines the microservices architecture cohesion as the sum of cohesion of all MS_c in the MSA.

$$Cohesion \;=\; \sum^{\forall c \in MSA} C(c) \tag{5}$$

Reuse. We computed microservices candidate reuse considering the relationships between the microservices candidate and the user of the legacy system (e.g, calling the API or user interface).

In this order, we propose to combine static and dynamic analysis to observe the level of reuse of a microservice within the microservices architecture. In the dynamic analysis, each microservices candidate is reusable when it is directly called by a user. This concept is captured by the *mdu* function (**m**icroservice **d**irectly called by the **u**ser). *mdu* function considers the system executions that allow identifying dynamic calls between vertices, including start points by the user.

$$r(M) \;=\; \begin{cases} 1, if \;\; \sum^{v_i \in M \wedge v_j \notin M} sc(v_j, v_i) + mdu(M) > 1 \\ 0, \; otherwise \end{cases} \tag{6}$$

In order to measure the reuse associated with each microservice, we defined the Equation 6. The goal of the equation captures the expectation that each microservice is useful for other microservices in the architecture or directly by the user. Whenever a microservice candidate is reused at least twice, the microservice candidate indicates an adequated reuse level.

The reuse of a microservices architecture is defined in Equation 7 when $|MSA|$ is the number of microservices. In addition, the property $0 < Reuse < 1$ is valid. The Equation 7 assume the value 1 when all microservices are used at least twice by other microservice or the user.

$$Reuse \;=\; \frac{\sum^{\forall M \in MSA} r(M)}{|MSA|} \tag{7}$$

Feature Modularization. We propose an strategy to indicate the responsibility of microservices candidates based on the features associated with executions of the target system. This information is provided by the user of `toMicroservices` approach. Basically, the user provides a list of features names

accessible via an interface (e.g, Rest API) of the legacy system under analysis. In addition, each feature label is associated with a part of an execution case (e.g, an interaction case) indicated by the user.

Consequently, `toMicroservices` performs the traceability between features labels and vertices (i.e, methods). This traceability is made during the execution of the legacy system responsible for implementing them. We used the vertices labeled to recommend feature modularization in the microservices candidates with fine granularity and limited responsibility.

In this order, the notion of predominant feature was created to indicate the occurrence of the feature that most occurs in the vertices (methods) associated with a microservice candidate. This notion of predominant feature is used to minimize the amount of features per microservices. Equation 8 defines the predominant feature (pf function) of a MS_c, where F_{M_c} is a set of occurrence by features in a MS_c.

$$pf(MS_c) \ = \ max \ _{\forall k \in F_{MS_c}} \{k\} \tag{8}$$

$$f(MS_c) \ = \ \frac{pf(MS_c)}{\sum^{\forall k \in F_{MS_c}} k} \tag{9}$$

Thus, the feature modularization of a microservices candidate was defined in Equation 9, that is, a measure of the number of occurrences of the most common features divided by the sum of all features occurrences within a microservices candidate.

Regarding the feature modularization in the proposed microservices architecture, Equation 9 introduced the feature per microservices. This equation avoids the fact that each microservice candidate has largely different features.

Equation 10 shows the measurement of features requirement where $FMSA$ is the set of different predominant features in the MSA. The division in Equation 10 of FMSA cardinality and MSA cardinality is to avoid a separation of the same feature by different microservices candidates. In addition, the occurrences of the predominant feature in each microservice candidate are summed.

$$F \ = \ \sum_{\forall c \in MSA} f(c) \ + \ \frac{|FMSA|}{|MSA|} \tag{10}$$

Network Overhead. A potential problem is the generated network overhead from the extracted microservices, possibly affecting negatively non-functional requirements as performance. In order to minimize that problem, we created a heuristic to predict the network overhead. The heuristic uses the size of the objects and primitive types assigned as parameters between methods during the execution of the legacy system.

The network overhead measurement is showed in Equation 11 where the function $P(v_j)$ return the set of arguments used in a execution of the method

v_j. The function sizeOf(p, m) is the size of the p-th parameter in the m-th call from v_i to v_j.

$$overhead(v_i, v_j, m) = \sum^{\forall p \in P(v_j)} sizeOf(p, m) \tag{11}$$

$$dt((v_i, v_j)) \;=\; max_{m=1}^{m=dc(v_i,v_j)}(overhead(v_i, v_j, m)) \tag{12}$$

Thus, data traffic function (dt) is computed as shown in Equation 12, where dc function is the total of calls from method v_i to method v_j in execution time.

The network overhead in a proposed MSA set is defined in Equation 13. In summary, network overhead is the sum of the sizes of the trafficked data to each MS_c.

$$\begin{aligned} O(MS_k) &= \sum^{\forall v_i \in MS_k \wedge \forall v_j \notin MS_k} dt((v_i, v_j)) \\ Overhead \;&=\; \sum^{\forall MS_k \in MSA} O(MS_k) \end{aligned} \tag{13}$$