A Cyclomatic Complexity Generalization for a Composite Service

Rupinder Pal Singh and Hardeep Singh

***Abstract*—Service composition is an important software development activity in the various phases of a service-oriented system. Developers would be keen to gauge the maintainability of the services they compose from the services available in a system. Complexity is widely acknowledged as a predictor of maintainability. McCabe's cyclomatic complexity is accepted as a reliable metric for measuring complexity. This paper explains usefulness of a result from McCabe's work in computing cyclomatic complexity of composite modules or components. It suggests improvements to an existing formal model of service-oriented system. It then applies the McCabe's result to define recursively a cyclomatic complexity generalization for a composite service.**

***Index Terms*—Composite service, cyclomatic complexity, metric, service-oriented architecture.**

I. Introduction

A *Service-oriented system, SOA-based system or SOA solution* is a distributed software system that is based on the architectural style *service-oriented architecture* (SOA), where systems consist of service users and service providers [1], [2]. The computing paradigm that utilizes SOA as the architectural style for developing service-oriented software is called service-oriented computing (SOC) [3]. An *SOA ecosystem* is an environment encompassing one or more *social structure(s)* and SOA-based system(s) that interact together to enable effective business solutions. A social structure is defined as a nexus of relationships amongst people brought together for a specific purpose.

SOA can be understood in terms of two basic concepts: layers and binding. Fig. 1 shows the SOA layers or the SOA stack [3]-[6]. In static binding (Fig. 2) the service requesters are bound to provided services at design time, whereas in the case of dynamic, run-time scenario (Fig. 3), service requesters dynamically discover, select the requisite services from a registry, and bind thereof to selected services.

Service composition is an important software development activity in the various phases of a service-oriented system [7]. Developers would be keen to gauge the maintainability of the services they compose from the services available in a system [8], [9]. Complexity can be seen as degree of difficulty in understanding the structure of design artifacts; or the amount of the internal work performed by a design artifact [10]-[12]. Complexity is an important structural or design characteristic besides size, coupling and cohesion. Structural properties represent internal quality and they are correlated to external quality characteristics such as *maintainability*, reliability, and performance. It has been widely accepted that high quality software should exhibit low complexity [13]-[17]. McCabe's cyclomatic complexity (MCC) is widely accepted as a reliable design metric for measuring complexity [18], [19]. This paper explains usefulness of a result from McCabe's work in computing cyclomatic complexity of composite modules or components. It suggests improvements to an existing formal model of service-oriented. It then applies McCabe's result to define recursively a cyclomatic complexity generalization for a composite service.

The remaining paper is structured as follows. Section II discusses an important formal model for service-oriented system and the MCC metric in brief. Section III covers related work. Section IV suggests improvements to an existing SOS formal model. Section V presents our recursive metric for composite service. Section VI presents a brief discussion and Section VII concludes and discusses future research possibilities.

II. Formal Models and Metrics

A. The Perepletchikov-Ryan-Frampton-Schmidt Model

We present here briefly the elements of the Perepletchikov-Ryan-Frampton-Schmidt model [16], [20]-[22]. There are many other models, metrics and measurement work [5], [13], [23]-[29]. In the general case, a service-oriented system, *SOS*, is formally defined as: $SOS = < SI, BPS, C, I, P, H, R >$, where *SI* is the set of all service interfaces in the system; *BPS* is the set of all business process scripts; *C* is the set of all object-oriented (OO) classes; I is the set of all OO interfaces; P is the set of all procedural packages; and H is the set of all package headers. Generically, the elements of these sets are called service implementation elements, each denoted as *e*.

Given a system, *SYS*, a service *s* can be defined as: $s = < si_s,\ BPS_s,\ C_s,\ I_s,\ P_s, H_s, R_s >$ is a service of *SYS* if and only if $si_s \in SI \wedge \{(BPS_s \subseteq BPS \wedge\ C_s \subseteq C \wedge I_s \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H) \wedge\ (BPS_s \cup\ \ C_s \cup\ I_s \cup P_s \cup H_s <> s) \wedge R_s \subseteq R\}$. The <> symbol represents *service membership*. A service boundary is logical rather than physical. The model proposes that we need to examine the possible *call paths* in response to invocations of service operations via the service interface in order to determine whether an element is a member of a service.si_s is a singleton set since a service *s* will have just one service interface si_s. R is the set of overall static coupling relationships (design-time and inter-modular) of *SYS*, i.e., $R \subseteq R_p \subseteq\ \ E \times E$, where E is the set of all service implementation elements *e's*, i.e., $E = SI \cup BPS \cup C \cup I \cup$

Manuscript received December 7, 2020; revised February 28, 2021.
R. P. Singh is with I. K. G. Punjab Technical University and CSE, GIMET, Amritsar, India (e-mail: rupi.pal@gmail.com).
H. Singh is with Dept. of CS, Guru Nanak Dev University, Amritsar, India (e-mail: hardeep.dcse@gndu.ac.in).

DOI: 10.7763/IJCTE.2021.V13.1289

$P \cup H$. R_p is the set of all common and possible relationships on $E \times E$. The static coupling relationships of service s, R_s, can be categorized as:

Interface to implementation relationships, $IIR(s) = \{(si, e): si = si_s \wedge e \in (BPS_s \cup C_s \cup P_s)\}$ (1)

Internal service relationships, $ISR(s) = \{(e_1, e_2): e_1, e_2 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)\}$ (2)

Incoming relationships, $IR(s) = \{(e_1, e_2): e_1 \in (BPS - BPS_s \cup C - C_s \cup I - I_s \cup P - P_s \cup H - H_s) \wedge e_2 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)\}$ (3)

Outgoing relationships, $OR(s) = \{(e_1, e_2): e_1 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s) \wedge e_2 \in (BPS - BPS_s \cup C - C_s \cup I - I_s \cup P - P_s \cup H - H_s)\}$ (4)

Service incoming relationships, $SIR(s) = \{(e, si): e \in (BPS - BPS_s \cup C - C_s \cup P - P_s) \wedge si = si_s\}$ (5)

Service outgoing relationships, $SOR(s) = \{(e, si): e \in (BPS_s \cup C_s \cup P_s) \wedge si \neq si_s\}$ (6)

$R_s = IIR(s) \cup ISR(s) \cup IR(s) \cup OR(s) \cup SIR(s) \cup SOR(s)$ (7)

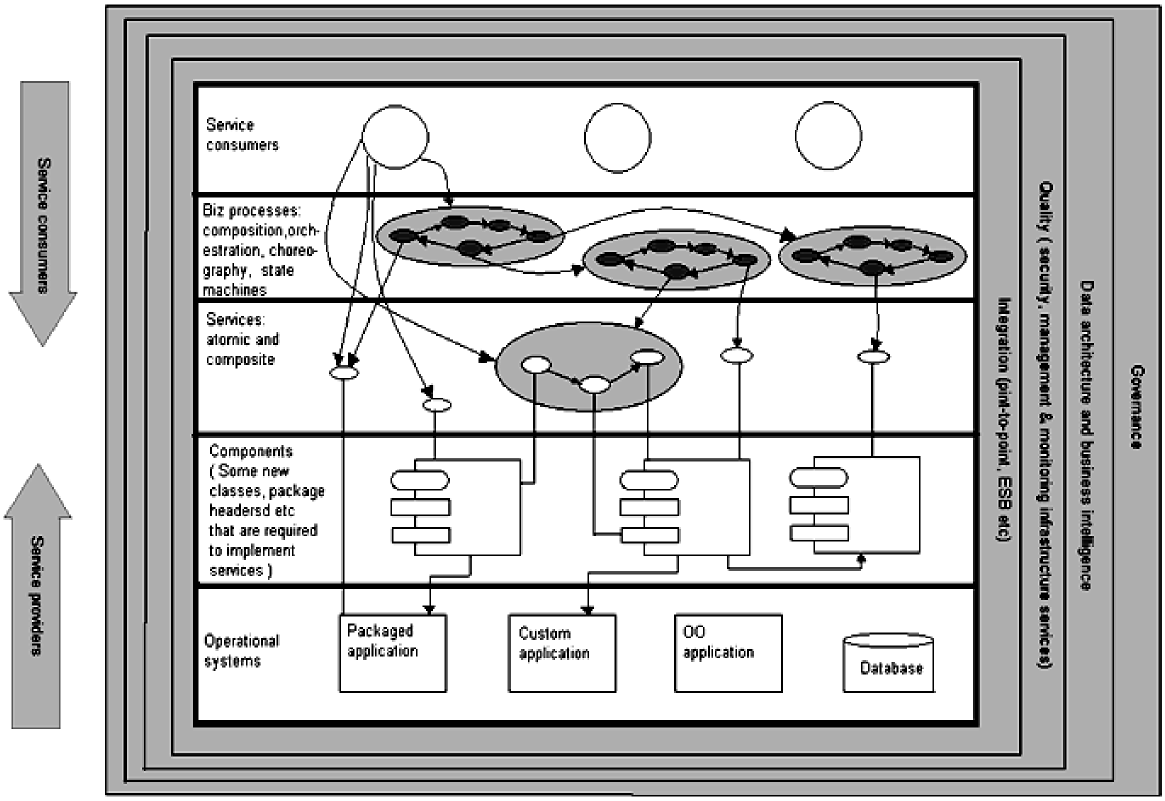

Fig. 1. SOA layers.

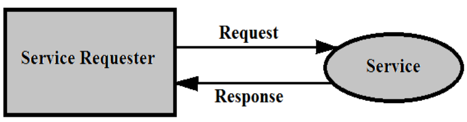

Fig. 2. Static binding.

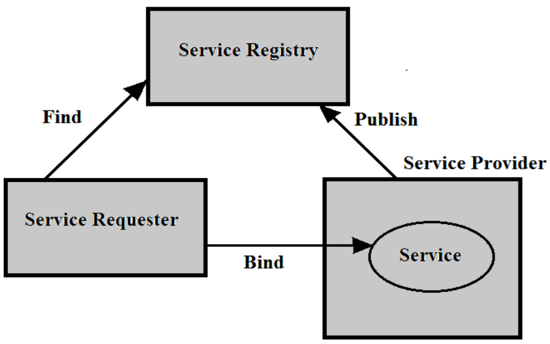

Fig. 3. Dynamic binding.

B. *McCabe's Cyclomatic Complexity*

MCC can be briefly explained as follows [15]. Control flow graph is a directed graph with unique entry and exit points. Each node in the graph corresponds to a block of code in the program where the flow is sequential, and the edges correspond to the branches taken in the program. It is assumed that each node can be reached by the entry node and each node can reach the exit node. The cyclomatic complexity of a control flow graph (CFG), whether structured or unstructured, with p connected components is

$$C = e - n + 2p \tag{8}$$

For a CFG with single connected component,

$$C = e - n + 2 \tag{9}$$

Alternatively, if there are π simple predicates (a decision node with either of two outcomes or a condition), the cyclomatic complexity of the CFG, whether structured or unstructured, is

$$C = \pi + 1 \tag{10}$$

III. Related Work

MCC has had many applications, having been adapted to parallel programs [19], concurrent module network model [14] and embedded software [30]. Vasconcelos *et al.* [31] have adapted MCC to derive a complexity metric for what they call ISA (Information System Architecture). The work by Perepletchikov *et al*. treats complexity for service-oriented systems as an amalgamation of cohesion and coupling [16]. Cardoso's work for business process workflows borrows some ideas from McCabe's cyclomatic complexity [32], [33]. However, Cardoso's work seems to apply to generic business processes, not to those related to service compositions like BPEL workflows. Mao [26] describes MCC for composite services specified using Petri-Nets. Gruhn and Laue [13] discuss, besides Cardoso's work, issues involved in defining MCC for a business process workflow in the classic fashion (edges-nodes+2). They do mention that nested structures (e.g., modules/composing-services) should contribute to greater overall complexity. However, none of Cardoso, Mao and Gruhn-Laue recursively take into account the underlying modules and components. McCabe does suggest a method to calculate the complexity of a collection of programs, particularly a hierarchical nest of subroutines [15]. It is this method that we employ to define a recursive definition of cyclomatic complexity for a composite service.

Hall and Preiser [14] argue that MCC as adapted to network of modules should take into account complexities of individual modules at nodes. Hall and Preiser do suggest a metric similar to ours but they do not refer to the McCabe's result (MR, as discussed in the Section V) and as a result do not suggest an exact cyclomatic complexity metric for network of modules. We do not find any report on recursive generalization of cyclomatic complexity for a composite service. Though our previous work [34] did propose a cyclomatic complexity metric for a service that reflects a few ideas we present here, it does not directly treat composite service and presents no derivation as we do here. In particular, the work suggests that cyclomatic complexity of a service should be sum of cyclomatic complexities of all its operations, treating the whole service as a graph of control flow graphs of operations as disconnected components. Our present work fulfills the needs explained here.

IV. Formal Model Improvements

Our metric is applicable to service compositions created using standard programming frameworks (e.g., Java Web Services). A typical scenario is shown in Fig. 4 [35]. Hansen [35] calls such applications "enterprise-quality SOA applications." A typical service implementation element of the composition is shown in the Listing 1. Such compositions can be modeled as CFGs since control flow analysis (CFA) is suitable for analyzing structured and object-oriented programs [36]. However, it is even possible to apply, in a restricted manner, as explained in the Section VI, the metric to business process compositions obtained using service composition engines like BPEL.

```
public abstract class ShopperImp {

public static ShopperImpnewShopperImp(Store src) {
if (src == null) {
throw new IllegalArgumentException("src may not be
null.");
}
switch (src) {
case YAHOO:
return
newYahooShopperImp(ShopperCredentials.getYahooA
ppID());
case EBAY:
return new
EBayShopperImp(EBayShopperImp.EBAY_PRODUC
TION_SERVER,
EBayShopperImp.SITE_ID_US,
ShopperCredentials.getEBayAppID());
case AMAZON:
return new
AmazonShopperImp(ShopperCredentials.getAmazonA
ccessKeyID());
default:
throw new RuntimeException("Unknown source: " +
src.getName());
}
}
```

Listing 1. The ShopperImp.newShopperImp() Factory Method

A composite service itself is a recursive composition of *composing* services (atomic, composite or both), components and standard programming nodes. Any metric for a composite service would need to take that into account. The complexity metric that we intend to generalize for a composite service, MCC, is essentially CFG-based. We needed an SOS model that is graph-based, structure- and behavior-based in terms of implementation elements and thus would allow us to delineate CFGs of implementation elements in a bottom-up manner. We found the Perepletchikov-Ryan-Frampton-Schmidt model (sub-section II.A) a suitable choice. However, some issues that we identify in relation to the model are:

a) The logical boundary of a service is not clearly defined. Given the graph union of sets *CS*es, where a *CS* itself is a graph union of all *invocation/call sequences* (each denoted as *cs*) possible for a service operation across elements (*or* module*s, e's),* the model defines the set of elements across this graph union to be the logical boundary of the service. Symbolically, this set is $BPS_s \cup C_s \cup I_s \cup P_s \cup H_s$. The model restricts the elements of this set to "reachable" elements, excluding *called/invoked* elements participating in *OR*(*s*). The model excludes them for atomic services ($SOR(s) \cup OR(s) = \Phi$) but includes them for composite services ($SOR(s) \cup OR(s) \neq \Phi$). This is inconsistent. It appears that the model has not clearly distinguished among the concepts of abstract sequential control flow (as represented by a CFG) of an executable artifact, invocations/calls the artifact would make as function calls (e.g., recursive, static method calls etc.), invocations/calls on injected dependencies (also an *e*) like dynamic web components, the nested calls those calls might make in turn (again, on *called/invoked* elements participating in the respective OR(e)'s of those elements, whether functions or injected dependencies) and calls to composing-service operations.

b) An atomic service is not clearly defined. The definition given is: A service *s* with *SOR*(s) ⋃ *OR*(s) = Φ is called an atomic service. It misses requiring that the set BPS_s be a null set. BPSes are, as also assumed in this model, executable composite services. As another gap, consider a CDI-style bean that is defined as a JAX-RS root resource class as in the Listing 2 would be exposed as an atomic service. The element e1, the root resource class, shows dependency on another element e2, a container-managed component, *MyOtherCdiBean*. The element e2 is a reusable component and could be injected anywhere else as well in the global namespace of the web server. This dependency is clearly an outgoing relationship and thus an element of *OR*(*s*).

c) The standard definition of an atomic service, as follows, does not necessarily require *OR*(*s*) to be a null set: An atomic service is a well-defined, self-contained function that does not depend on the context or state of other services [37], [38]. Defining atomic services clearly would make the model more in line with the widely accepted layering shown in Fig. 1 and the ISO/IEC 18384-1-3 standard [1]; it is clear atomic services are basic blocks whereas composite services can appear in the higher business process layer of an SOS as well. The definition of SIR(s) does not include static incoming relationships from composite services other than BPS. For example, the kind of composite service we introduced in the beginning of this section (Fig. 4) is not a bps.

d) A composite service or an atomic service itself has not been included as an element of either a system SOS or a service *s*. If services are allowed to be composed from atomic and other composite services, those composing services themselves become elements of the SOS. The ISO/IEC 18384-1-3 standard [1] specifies that any service, whether atomic or composite, would itself be an element of SOS.

The above points analyzed together lead us to conclude:

i. The logical boundary of any public service operation should be the union of the CFG of its main thread of execution and CFGs of all its *explicit* child threads (*if any*). Each such CFG constitutes a separate connected component. Function- and injected-dependency calls (synchronous, asynchronous, global, static method calls, recursive or any valid combination thereof) and composing-service calls will each be represented as a node in the CFGs and thus be part of the logical boundary. The executions of such calls are not part of the logical boundary. All possible executions of a call constitute separate CFG. This concept is explained later in the Section V using McCabe's result. The logical boundary of a service should be the graph union of all such logical boundaries of its operations. If there is a call *c1* to an operation *o1* of an element *e* and another call *c2* to a different operation *o2* of *e*, each such call is a node. If there is another call *c3* to the same operation *o1* of the same element *e*, it will also be a separate node.

ii. An *SOS* should be defined as *SOS* =<*SI*, *C*, *I*, *P*, *H*, *A*, *CPS*, *R*>, where A denotes all atomic services and *CPS* denotes all composite services in the system. *CPS* will include composite services created on top of service composition engines as also those created on top of application programming frameworks.

We can now define a service recursively as follows.

Given a service-oriented system, *SYS*, a service *s* can be defined as:

a) $s = < si_s, C_s, I_s, P_s, H_s, f_s, R_s >$ is a service of *SYS* if and only if $si_s \in SI \wedge \{(C_s \subseteq C \wedge I_s \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H) \wedge (C_s \cup I_s \cup P_s \cup H_s) = D(f_s) \wedge (R_s \subseteq R)\}$.

```
@Path("/cdibean")
public class CdiBeanResource {
@Inject MyOtherCdiBean bean; // CDI injected bean
@GET
@Produces("text/plain")
public String getIt() {
return bean.getIt();  }
}
```

Listing 2. A JAX-RS root resource class.

f_s is the logical boundary of the service *s*. Only elements that are either inlined (such as header files in C++) to the logical boundary of a service or *used* (such as OO interfaces) by elements that are in the logical boundary and *not reused* anywhere else except within a service can be said to exclusively belong to the service. These elements are extracted by *D*( ) as the *set D*(f_s). Such a service is called an atomic service.

b) $s = < si_s, C_s, I_s, P_s, H_s, A_s, CPS_s, f_s, R_s >$ is also a service of *SYS* if and only if $si_s \in SI \wedge \{(C_s \subseteq C \wedge I_s \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H \wedge A_s \subseteq A \wedge CPS_s \subseteq CPS) \wedge (C_s \cup I_s \cup P_s \cup H_s) = D(f_s) \wedge (R_s \subseteq R)\}$. Such a service is called a composite service.

$R \subseteq E \times E$ where E is the set of all service implementation elements *e'*s, i.e., $E = SI \cup C \cup I \cup P \cup H \cup A \cup CPS$. R is the set of all common and possible relationships of an *SOS*.

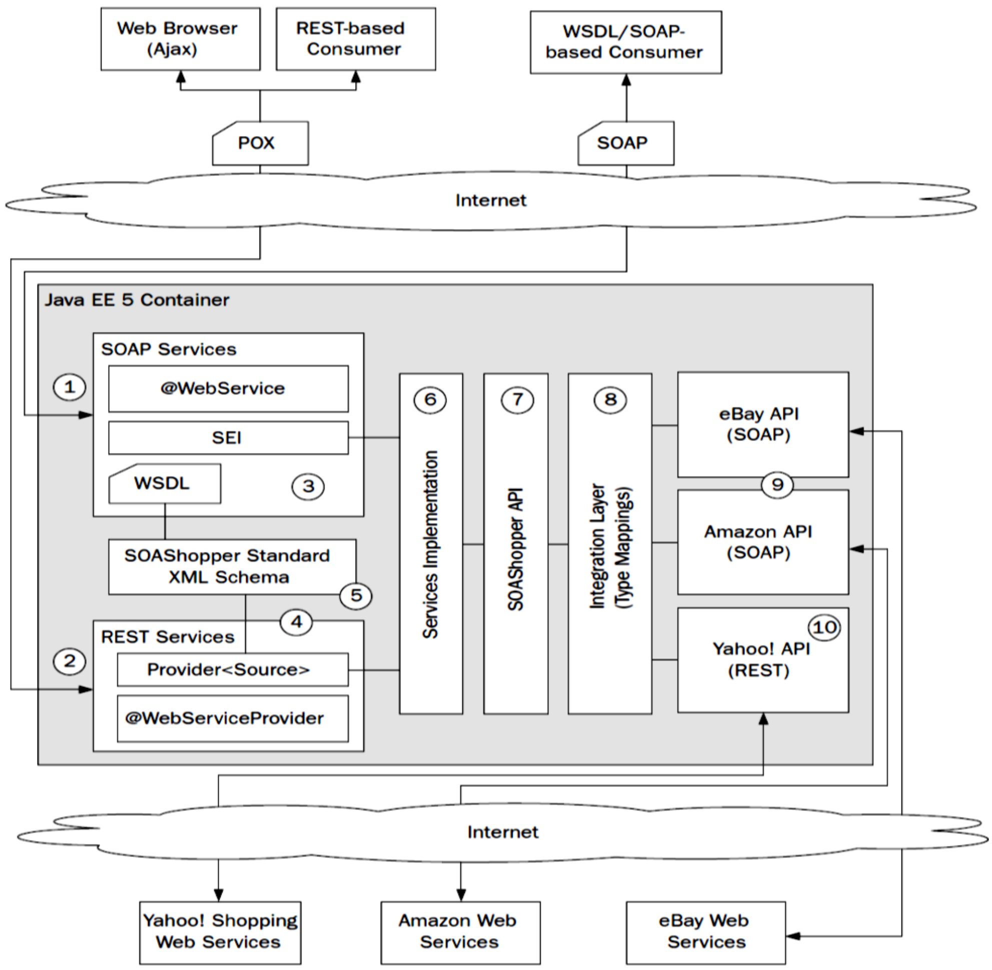

Fig. 4. SOAShopper architecture, an example of composition using standard programming frameworks.

V. Cyclomatic Complexity for a Composite Service

McCabe [15] argues that tracking the MCC of a program under development and keeping it low should help in modularization of the program and thus keep it testable and maintainable. More specifically, he explains that every structured program can be reduced to the CFG shown in the Fig. 5 by successively replacing its every control flow subgraph (that is, a subgraph with unique entry and exit nodes) with a single node. The CFG in the Fig. 5 has essential complexity (*ec*) of 1. Likewise, every unstructured CFG with *m* control subgraphs has essential complexity,

$$ec = C - m \tag{11}$$

where C is its MCC.

If all its control subgraphs are successively removed, replacing each with a single node, we get a fully unstructured CFG with essential complexity equal to its MCC.

$$ec = C - 0 = C \tag{12}$$

Fig. 5. The CFG with unit essential complexity.

The essential complexity of a graph indicates the extent to which it can be reduced. Each removed control graph can be implemented as a separate module. In other words, whether it is a structured or unstructured graph, the process of modularization involves reducing its MCC to a suitable complexity. One might still be interested in computing the complexity of the overall program (main program and its modules). The process of composition is a related but slightly different process. One starts with a main program of suitable complexity and as more and more nodes are implemented as interface invocations/calls to reusable modules or components, either available off-the-shelf or developed from scratch, the complexity of the overall program (main program and its modules) might need to be tracked too. Significantly, to compute the cyclomatic complexity of the overall program, McCabe presents a result [15]. He provides justification using an example as reproduced in Fig. 6. Suppose there is a main routine M that calls subroutines A and B. All three routines taken together are treated as one collection consisting of three connected components.

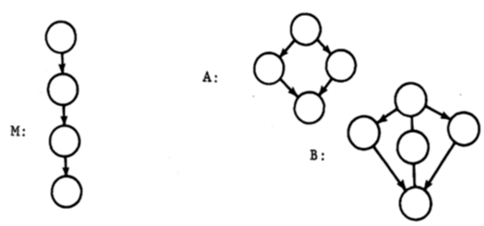

Fig. 6. McCabe's example.

The main routine maintains the abstract sequential control in the manner imposed by these CFGs. It does not transfer this control to any of the sub-routines. The main routine suspends (blocks) its abstract sequential control by storing the current program counter (PC) on a call stack. In other words, the main routine only transfers the machine control to a subroutine, which then starts its complete sequential flow till the end and then transfers back the machine control to the main routine. The main routine resumes its abstract sequential flow at the PC it blocked by retrieving it from the stack. If it is an asynchronous call, the main routine does not even suspend; the call is executed on a separate thread. This scenario applies to the situations where an operation of a service implementation element e or a composite service calls operations on some other composing components or services. Applying the formula (Eq. 8) for connected components to the example in Fig. 6 with p=3, the complexity C is,

$$C = e - n + 2p = 13 - 13 + 2\times 3 = 6 \quad (13)$$

Also,

$$C = C(M) + C(A) + C(B) = 2 + 2 + 2 = 6 \quad (14)$$

McCabe's Result (MR): In general, the complexity of a collection of k control graphs is equal to the summation of their individual complexities,

$$C(G) = e - n + 2p = \sum_1^k e_i - \sum_1^k n_i + 2k = \sum_i^k (e_i + n_i + 2) = \sum_i^k C_i \quad (15)$$

McCabe clarifies that the above result can be used to calculate the complexity of a collection of programs, particularly, such as a hierarchical nest of subroutines. For example, to compute the overall complexity of an operation of a composite service or component, that, in turn, calls some operations on other services or components, the cyclomatic complexities of CFGs of individual invoked operations of composing services or components are simply added to the cyclomatic complexity of the operation. In general, McCabe's result is applicable to a graph consisting of separate connected components.

We now recursively define a cyclomatic complexity generalization for composite services. For any thread of execution, for example, a thread of execution of a function-call or an injected dependency call, given that the nodes in its CFG are standard programming nodes, its cyclomatic complexity C_p is

$$C_p = edges - nodes + 2 \quad (16)$$

The CFG constitutes the logical boundary (as explained in the Section IV) of the thread of execution, p.

Next, we describe computation of the cyclomatic complexity of a multi-thread concurrency program cp. Developers might use such a concurrency in writing service implementation elements. We assume that developers use only standard programming nodes in writing these threads. The CFGs of the main thread and its explicit child threads are separate connected components. Applying MR,

$$C_{cp} = C_{mt} + \sum_l C_{ctl} \quad (17)$$

where C_P is the complexity of the complete program cp, C_{mt} is complexity of the main thread and C_{ctl} is complexity of l^{th} child thread. C_{mt} and C_{ctl} can be computed using the Eq. 16. The union of the CFGs of the main thread and the explicit child threads constitute the logical boundary of the program cp.

Next, let us treat a recursive function. We assume that the function is written using standard programming nodes only. For a recursive function, the cyclomatic complexity will just be the complexity of the CFG of the function. All nested recursive calls will be made to the same function.

$$C_r = C_{cf} \quad (18)$$

C_{cf} can be computed using the Eq. 16. The CFG constitutes the logical boundary of the recursive function, r.

Consider a generic software artifact encapsulating some functionality that is available via call/invocation: e.g., an operation of a service (e.g. an operation of a service endpoint class), an operation of an element e etc. Denote it as o. Suppose o, in turn, makes a number of dependency calls to *other similar software artifacts*. One or more calls to the same artifact will be treated as one outgoing static coupling. Each such coupling will be an element of $OR(o)$. Any other elements called by calls nested further are also similar software artifacts. Denote p^{th} operation called by a dependency call as *dop*. The cyclomatic complexity of the software artifact o is, applying MR,

$$C_o = C_{of} + \sum_p C_{dop} \quad (19)$$

where C_{of} is the complexity(computed using any combination applicable from the Eqs. 16-18) of the logical boundary of the operation o and C_{dop} is the complexity of the p^{th} dependency operation *dop* called from the logical boundary of o.

Consider a service implementation element e such that e is in $C \cup P$. The cyclomatic complexity of any of its public operations can be defined as follows. (Service interfaces SI and OO interfaces I do not have any control flow complexity; and package headers H do not have any stand-alone control flow complexity since they are supposed to contain only inline functions and a compiler will compile a package header along with some procedural package or class). For every operation *eo* of an implementation element e, there will be a logical boundary *across* standard programming nodes (e.g., if else) and dependency calls to i operations *deo* of some other *similar* elements e ($e \in C \cup P$ and any other elements called by calls nested further are also $e \in$ C $\cup$ P). Each dependency call (e.g., a function-call, injected-dependency call etc.) to an e's operation is treated as a node in the logical boundary. (The executions of such calls constitute separate logical boundaries.) One or more calls to the same operation will be treated as one outgoing static coupling. Each such coupling will be an element of $OR(eo)$.The complexity of the logical boundary (we use f to denote it) can be calculated using any combination applicable from the Eqs. 16-18 as C_{eof}. Let the complexity of an operation of a dependency that is called be denoted with C_{deo}. The total complexity of *eo* will be, applying Eq. 19,

$$C_{eo} = C_{eof} + \sum_i C_{deoi} \quad (20)$$

Consider an atomic service *as*. For each operation *aso* in the atomic service *as*, there will be a logical boundary *across* standard programming nodes (e.g. if else) and nested calls to operations *deo*'s of some other elements *e*. Each nested call (e.g., a function-call or injected-dependency call) to any other *e*'s operation is treated as a node. (As clarified earlier in this section, the executions of such calls constitute separate logical boundaries.) One or more calls to the same operation will be treated as one outgoing static coupling. Each such coupling will be an element of *OR* (*aso*). The complexity of the logical boundary is computed using any combination applicable from the Eqs. 16-18. Let this be denoted C_{asof}. Then, applying Eq. 19, the total complexity of each operation is computed as

$$C_{aso} = C_{asof} + \sum_q C_{deoq} \tag{21}$$

where *deoq* (using Eq. 20)is the complexity of the q^{th} element-operation called from the logical boundary of *as*.

For the atomic service *as*, each of the logical boundaries of its various operations form separate connected components. Applying MR,

$$C_{as} = \sum_j C_{asoj} \tag{22}$$

As we mentioned in the Section III, our previous work [34] reports this metric.

Consider a composite service *cps*. For each operation *cpso* of the service, there will be a logical boundary across *composing-service* operation calls (each treated as a node), dependency calls to operations *deo*'s on some *e*'s (*e* is not a service but can make nested calls to services) and standard programming nodes (e.g. if else). Each call to an *e*'s operation is treated as a node. (The executions of such calls constitute separate logical boundaries.) One or more calls to the same operation will be treated as one outgoing static coupling. Each such coupling will be an element of *OR* (*cps*). The cyclomatic complexity of the logical boundary is computed using any combination applicable from the Eqs. 16-18 as C_{cpsof}. Let the complexity of a k^{th} operation invoked on a *composing* service (it can be either atomic or composite) be denoted by C_{cosok}. Let the complexity a u^{th} operation invoked via a dependency call be C_{deo} (using Eq. 19). Applying Eq. 19, the total complexity of the composite service operation *cpso* is

$$C_{cpso} = C_{cpsof} + \sum_k C_{cosok} + \sum_u C_{deou} \tag{23}$$

For the composite service *cps*, each of the logical boundaries of its various operations form separate connected components. Applying MR,

$$C_{cps} = \sum_v C_{cpsov} \tag{24}$$

C_{cpsov} is the complexity of v^{th} operation. C_{cps} denotes the cyclomatic complexity of the composite service *cps*.

VI. Discussion

It is even possible to apply, in a restricted manner, the metric to business process compositions as achieved using BPEL [39], [40] provided there are no concurrent/parallel elements (like <flow>, parallel <for-each>), synchronizing dependencies (defined by <link> node) and external one-way events (the <invoke> activity should not allow a business process to invoke a one-way call on a port Type offered by a partner and there should be no <onAlarm> events). A <scope> node should be treated as a single node in the logical boundary of the composition.

With these assumptions, the business process flow graph is the same as the logical boundary of the business process. Consider a business process graph for a composite service with a single operation as an example as in Fig. 7 [17]. Applying Eq. 23,

$$C_{cpso} = C_{cpsof} + \sum_k C_{cosok} \tag{25}$$

Using Eq. 16, the cyclomatic complexity for the logical boundary is 2. There are six nodes and six edges. So,

$$C_{cpsof} = 6 - 6 + 2 = 2 \tag{26}$$

Assume every operation corresponding to a service-operation invocation, *coso*, has complexity 2.

$$C_{cpso} = 2 + 2 + 2 + 2 = 8 \tag{27}$$

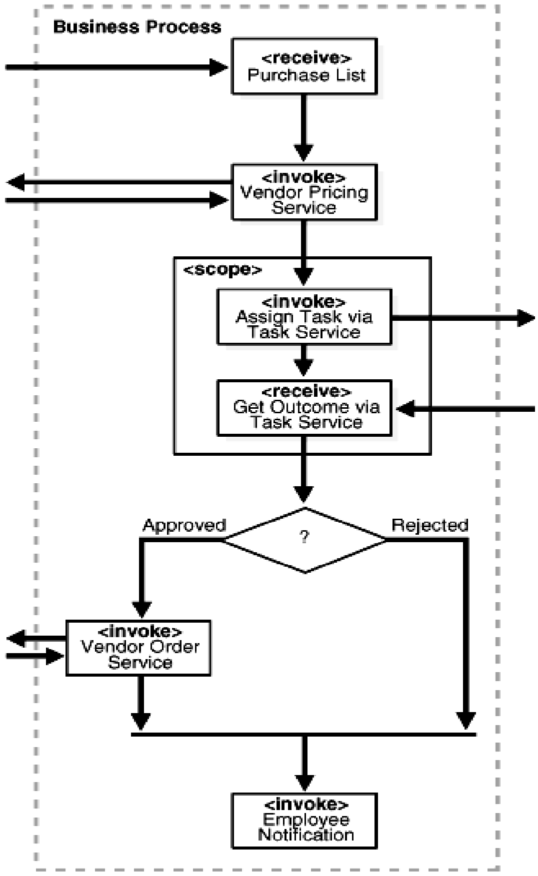

Fig. 7. BPEL workflow.

VII. Conclusion

This paper identifies and explains an exact cyclomatic complexity metric for composite components. Further, it presents a recursive definition of cyclomatic complexity metric for a composite service. Complexity is an important design predictor of maintainability and a comprehensive complexity metric as cyclomatic complexity will help developers in gauging the maintainability of composite services they compose.

Moreover, the method can be generally applied to any composite component or module. The paper also demonstrates initial work toward making fundamental improvements to a prominent model. In our future work, we intend to take forward these improvements, develop a comprehensive model and suggest more metrics.

Conflict of Interest

The authors declare no conflict of interest.

Author Contributions

R.P. Singh reformulated the model and worked out the metric. H. Singh guided the entire work and wrote the final version of the paper. Both authors had approved the final version.

References

[1] ISO/IEC, Information technology--Reference Architecture for Service Oriented Architecture (SOA RA), ISO/IEC 18384-1, 3 & 3. First edition 2016-06-01.

[2] OASIS. Reference Architecture Foundation for Service Oriented Architecture Version 1.0, 04 December 2012. OASIS Standard. [Online]. Available: http://docs.oasis-open.org/soa-rm/soa-ra/v1.0/cs01/soa-ra-v1.0-cs01.html

[3] B. Portier. (2007). SOA Terminology overview, Part1: Service, architecture, governance, and business terms. [Online]. Available: http://www.ibm.com/developerworks/webservices/library/ws-soa-term1/?S_TACT=105AGX04&s

[4] C. Emig *et al.* The SOA's Layers. [Online]. Available: http://www.cm-tm.uka.de/CM-Web/07.Publikationen/%5BEL+06%5D_The_SOAs_Layers.pdf

[5] D. Rud, A. Schmietendorf, and R. Dumke, "Resource metrics for service-oriented infrastructures," in *Proc. SEMSOA 2007*, 2007, pp. 90-98.

[6] D. Russell and J. Xu, "Service oriented architecture in the provision of military capability," in *Proc. UK e-Science All Hands Meeting*, 2007.

[7] Q. Z. Sheng *et al.*, "Web services composition: A decade's overview," *Information Sciences*, vol. 280, pp. 218–238, 2014.

[8] D. Ameller, "A survey on quality attributes in service-based systems," *Software Qual. J.*, vol. 24, pp. 271–299, 2016.

[9] J. Bogner *et al.*, "Exploring maintainability assurance research for service- and microservice-based systems: Directions and differences," in *Joint Post- Proc. of the First and Second International Conference on Microservices.*

[10] J. Bogner *et al.*, "On the impact of service-oriented patterns on software evolvability: A controlled experiment and metric-based analysis," *Peer J. Computer Science*, vol. 5, p. 213, 2019.

[11] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *Proc. 7th International Conference on Quality Software*, 2007.

[12] S. Rangarajan *et al.*, "Web service QoS prediction using improved software source code metrics," *PloS one*, vol. 15, no. 1, p. e0226867, 2020.

[13] V. Gruhn and R. Laue. Complexity metrics for business process models. [Online]. Available: http://ebus.informatik.uni-leipzig.de/~laue/papers/metriken.pdf

[14] N. R. Hall and S. Preiser, "Combined network complexity measures," *IBM. J. R. D.*, vol. 28, no. 1, Jan. 1984.

[15] T. J. McCabe, "A complexity measure," *IEEE Transactions on Software Engineering*, vol. 2, no. 4, 1976.

[16] M. Perepletchikov, "Software design metrics for predicting maintainabaility of service-oriented software," Ph.D. thesis, RMIT Univ., Melbourne, Feb. 2009.

[17] S. VanderWiel, D. Nathanson, and D. J. Lilja, "Performance and program complexity in contemporary network-based parallel computing systems," Technical Report No. HPPC-96-02, March 1996, University of Minnesota.

[18] G. Polančič and B. Cegnar, "Complexity metrics for process models – A systematic literature review," *Computer Standards & Interfaces*, vol. 12, p. 3, 2016.

[19] S. VanderWiel, D. Nathanson, and D. J. Lilja, "Performance and program complexity in contemporary network-based parallel computing systems," Technical Report No. HPPC-96-02, March 1996, University of Minnesota.

[20] M. Perepletchikov, C. Ryan, and K. Frampton, "Cohesion metrics for predicting maintainability of service-oriented software," in *Proc. 7th International Conference on Quality Software*, 2007.

[21] M. Perepletchikov, C. Ryan, and K. Frampton, "Coupling metrics for predicting maintainability in service-oriented designs," in *Proc. 18th International Conference on Software Engineering*, 2007.

[22] M. Perepletchikov, C. Ryan, K. Frampton, and H. Schmidt, "Formalising service-oriented design," *Journal of Software*, vol. 3, no. 2, Feb. 2008.

[23] B. Gonen *et al.*, "Maintaining SOA systems of the future: How can ontological modeling help," in *Proc. the International Conference on Knowledge Engineering and Ontology Development*, 2014, pp. 376-381.

[24] A. Korostelev *et al.*, "Error detection in service-oriented distributed systems," in *Proc. IEEE Int. Conf. on DSN 2006*, 2006, vol. 2, pp. 278-282.

[25] Y. Liu and I. Traore, "Complexity measures for secure service-oriented software architectures," in *Proc. the 3rd IEEE International PROMISE Workshop*, 2007.

[26] C. Mao, "Control flow complexity metrics for petri net based web service composition," *Journal of Software*, vol. 5, no. 11, November 2010.

[27] D. Rud, A. Schmietendorf, and R. Dumke, "Product metrics for service-oriented infrastructures," in *Proc. 16th International Workshop on Software Measurement/DASMA Metrik Kongress 2006*,2006.

[28] T. Xu, K. Qian, and X. He, "Service oriented dynamic decoupling metrics," in *Proc. 2006 Intl. Conf. on Semantic Web and Web Services (SWWS' 06)*, June 26-29, 2006 WORLDCOMP' 06, Las Vegas, USA.

[29] W. Zhao, Y. Liu, J. Zhu, and H. Su, "Towards facilitating development of SOA application with design metrics," *Journal of Software*, vol. 5, no. 11, November 2010.

[30] G. Menkhaus and B. Andrich, "Metric suite for directing the failure mode analysis of emdedded software systems," in *Proc. 7th ICEIS'05*, 2005.

[31] A. S. Vasconcelos, P. Sousa, and J. Triblolet. Information system architectures: An enterprise engineering evaluation approach. [Online]. Available: http://www.inesc-id.pt/ficheiros/publicacoes/3543.pdf

[32] J. Cardoso, "Process control-flow complexity metric: An empirical validation," in *Proc. IEEE Intl. Conf. on Services Computing (IEEE SCC 06)*, Chicago, USA, Sept, 2006, pp 167-173.

[33] J. Cardoso, "Approaches to compute workflow complexity," in *Proc. Dastguhl Seminar 06291*, 2006.

[34] H. Singh and R. Singh, "On formal models & deriving metrics for service-oriented architecture," *Journal of Software*, vol. 5, no. 8, Aug 2010.

[35] M. D. Hansen, *SOA Using Java Web Services*, 2007 Pearson Education, Inc., USA.

[36] V. Garousi *et al.*, "Control flow analysis of UML 2.0 sequence diagrams," Carleton University TR SCE-05-09, September 2005.

[37] D.K. Barry and D. Dick, *Architectures, and Cloud Computing: The Savvy Manager's Guide*, 2nd Ed. Elsevier Inc., 2013.

[38] J. Ganci. (2006). *Patterns: SOA Foundation Service Creation Scenario*. [Online]. Available: http://www.redbooks.ibm.com/redbooks/pdfs/sg247240.pdf

[39] M. B. Juric. A Hands-on Introduction to BPEL. [Online]. Available: *https://www.oracle.com/technical-resources/articles/matjaz-bpel.html*

[40] OASIS. Web Services Business Process Execution Language Version 2.0, 11 April 2007. OASIS Standard. [Online]. Available: http://docs.oasis-open.org/wsbpel/2.0/OS/wsbpel-v2.0-OS.html#_Toc164738529

Copyright © 2021 by the authors. This is an open access article distributed under the Creative Commons Attribution License which permits unrestricted use, distribution, and reproduction in any medium, provided the original work is properly cited (CC BY 4.0).

Rupinder Pal Singh was born in Amritsar, India on the Aug 01, 1970. Singh earned a bachelor of Engg. degree in electrical engineering from Jabalpur Engineering College, Japalpur, India in 1993. Singh earned a master of Tech. degree in information technology from Guru Nanak Dev University, Amritsar, India in 2008. Singh was registered as a part-time Ph.D. scholar in computer science and engineering in 2012 and is in the process of submitting his thesis.

He is on a long study leave from his position of associate professor of computer science and engineering with GIMET, Amritsar, India. Previously, he held a position of associate professor in information technology with AIET, Faridkot, India. Besides the academic experience, he has an industry experience of about 12 years.

Prof. Singh contributes extensively to education-quality, co-curricular and extra-curricular committees of the institutes he is been affiliated to.

Hardeep Singh was born in Kapurthala, India on the Feb 16, 1963. Singh earned a Ph.D. in computer science and engineering from Guru Nanak Dev University, Amritsar, India in 2003.

He is currently a professor and the head of the Dept. of Computer Science in Guru Nanak Dev University. Previously, he held the positions of the head of the Dept. of Computer Science twice. He has several publications to his credit including six books.

Prof. Singh has held several administrative and academic positions in Guru Nanak Dev Uinversity, including chairs of Boards of Studies and Faculty of Computer Science and Engg, Dept. and Engg, and Tech. He was the dean of Faculty of Engg. and Tech. and the dean of Alumni. He held the positions of director of Capacity Enhancement Program and Placement.