Defining the Logical Boundary of a Service: An Improved Formal Model and Novel Metrics for Service-Oriented Systems

RUPINDER PAL SINGH[1]
HARDEEP SINGH[2]

[1]Research Scholar
I.K.G. Punjab Technical University, Kapurthala, India
E-mail: rupi.pal@gmail.com

[2]Deptt. of Computer Science
G.N.D. University, Amritsar, India
E-mail: hardeep.dcse@gndu.ac.in

Abstract— In the field of service-oriented systems, a service is considered as an artifact that has a logical representation. However, the logical boundary of a service is not clearly defined. In particular, it needs to be defined clearly at the design level. Without such a definition, it is not possible to delineate outgoing coupling of a service. It would be difficult to analyze overall static, inter-modular coupling of a service. Further, one cannot devise effective metrics for design characteristics like complexity, cohesion and coupling of a service. A definition that is both technology-agnostic and independent of the physical packaging of a service would be most suitable. This paper defines clearly the logical boundary of a service and makes other improvements to a generic formal model. Thus, it presents a comprehensive formal model that leads to novel metrics and helps in explaining microservices architecture as a special case.

Index Terms— Service-Oriented System, Service-Oriented Architecture, Formal Model, Metrics, Logical Boundary

(Received October 1st, 2020 / Accepted November 11st, 2020)

1. 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 [23, 35]. The computing paradigm that utilizes SOA as the architectural style for developing service-oriented software is called service-oriented computing (SOC) [44]. 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 [13][44][47][49]. 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.

In the field of service-oriented systems, a service is considered as an artifact that has a logical representation.

The stress is on identifying a service by its network-publishable interface. While it is important to maintain this essential black-box user view of a service, it is not a restriction at the design-level. However, a clear design-level definition of the logical boundary of a service is not available. Without such a definition, it is not possible to delineate outgoing coupling of a service. It would be difficult to analyze overall static, inter-modular coupling of a service in terms of various types of coupling. Further, one cannot devise effective metrics for design characteristics like complexity, cohesion and coupling of a service. A definition that is both technology-agnostic and independent of the physical packaging of a service would be most suitable. This paper defines clearly the logical boundary of a service and makes other improvements to a generic formal model. Thus, it presents a comprehensive formal model that leads to novel metrics and helps in explaining microservices architecture as a special case.

The remaining paper is structured as follows. Section 2 discusses the related work and establishes the need of our work. Section 3 provides theoretical ground for our work. Section 4 presents a heuristic argument leading to the definition of logical boundary of a service, gives the definition and explains the improved model. Section 5 defines metrics using the improved model. Section 6 concludes and discusses future research possibilities.

2. Related Work

Except for the Perepletchikov-Ryan-Frampton-Schmidt model (explained in the Section 4) and the SCA (Service Component Architecture) implementation paradigm of SOA, we found not much in the literature on models and metrics for service-oriented systems that could be considered to define the logical boundary of a service [17][18][27][29][30][47][48][54][55].

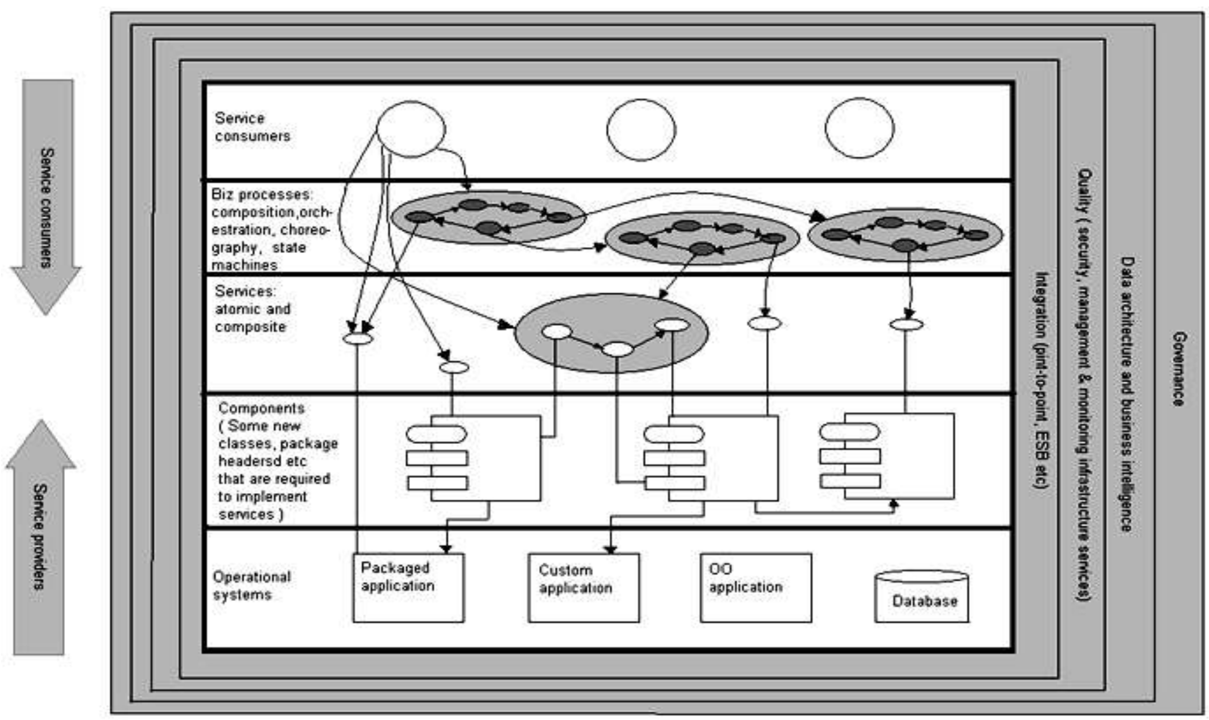

Fig. 1. The SOA Layers.

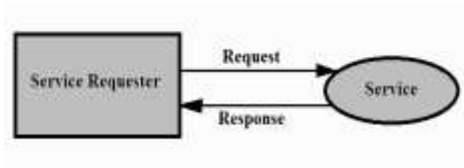

Fig. 2. Static binding.

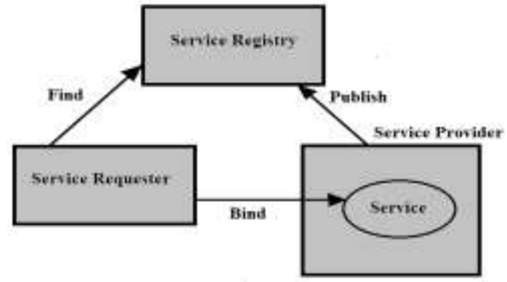

Fig. 3. Dynamic binding.

The logical boundary of a service is not clearly defined in the Perepletchikov-Ryan-Frampton-Schmidt model and we discuss this in Section 4. The SOA implementation paradigm, SCA, defines publically addressable services as *composites*. A composite is a unit of deployment. An incoming coupling can enter a composite only through one of the points on it called services. Each service is typed by an interface. An outgoing coupling can exit the composite through one of the points on it called references. Each reference is typed by an interface. Components within a composite are configured instances of a component's implementation. However, components cannot directly access any component or composites outside of the composite that deploys them. Nor can any composite or component not deployed by the composite can access any component within it directly [51]. Clearly, the graph connecting the components and the interfaces is used to define the logical boundary of a composite. Since component instances are configured (via XML scripts) within a composite, an SCA runtime cannot assign a component instance dynamically to any thread other than that requesting a service from the composite.

Guidi and Lucchi [19] define a service as a tuple with an element called internal process that should express service functionality using some formalism. They do not delve further into it and do not ground that element in terms of logical boundary. Massuthe et al. [31] define a service with the need to specify execution of its operations as per some internal control structure.

3. Theoretical Foundation for the Improved Formal Model

Some well-established ideas support our choice of control flow graphs (CFG) to delineate the logical boundary of a service. The earliest support for our approach emanates from the work of Dijkstra who reported the "THE" operating system as a society of abstract sequential processes organized as a hierarchy of levels [11]. He summarizes that the work of his team shows that the logical soundness of such a multiprogammimg system can be proved a priori and its implementation can admit exhaustive testing. His ideas were implemented as function-call hierarchy (call/invoke hierarchy) in most operating systems [20, 52]. Parnas cites Dijkstra's work frequently [36-37]. The gist of Parnas's work related to modular design is that there are forms of hierarchy other than the one reported for the "THE" operating system ("gives work to" hierarchy) and the function-call hierarchy, and that modular hierarchy is not necessarily either of these two hierarchies. It should be called "uses" hierarchy and is mainly decided at design-time. He asserts that defining an application program in the manner of "flowchart" or "chains of data transforming components" (as in gives-work and invoke hierarchies) could be an equivalent runtime representation but not a design-time representation. He seems to stress that "uses" modular hierarchy is design-time and it plays significant role in viewing software as a *family* of programs. It is apparent that this line of thinking has had much influence on the way software application systems (including service-oriented systems) were viewed later in the research domain and practice. It seems that in all these developments the need to delineate in a modular hierarchy the logical boundary of an application program as one abstract sequential process was not adequately emphasized. All the same, besides Dijkstra's work, there are a few other studies and ideas that support this need.

Pressman describes application software as consisting of standalone programs that solve a specific business need [45]. So, if we generically consider software to be a family of programs, the boundary of one application program should be discernible. The concept of transaction in database systems is an important heuristic for our approach. A transaction constitutes a logical boundary to a set of database access operations such that they leave the database in a consistent state and they do not conflict with other sets of database access operations. The way transaction serves as a logical unit is by imposing an abstract sequence on the operations within it. The sequential flow is abstract since every transaction has its own flow and the database system implements those transactions, not the underlying operating systems or machines directly.

The work by Broy[8] emphasizes that to correctly compose large, modular and hierarchical systems from components, merely specifying syntactic interfaces (function signature and parameters along with types) of a component is not enough; its black-box I/O behavior needs to be formally specified by a logical function between input channel(s) and output channel(a channel is the identifier for an infinite timed-stream of messages). He also shows that such functions are state machines.

Ravindran's work on dynamic real-time distributed systems [46] is relevant. He defines a software sub-system of such systems as a set of application *programs*, a set of devices (sensors and actuators), a communication graph of application programs and devices, and a set of paths. The connectivity of a path is the graph of application programs and devices that belong to the path. A path always has a *root* node (i.e., the beginning of the path) and a *sink* node (i.e., the end of the path). The root node of the path is the only node in the path that does not have an incoming edge from any other application programs or devices that belong to the path. The sink node of the path is the only node in the path that does not have an outgoing edge to any other application programs or devices that belong to the path. Michaloski et al. employ ideas similar to those described by Dijkstra to describe a concurrent hierarchical robot system. The application uses virtual control loops—akin to cyclic abstract sequential processes used by Dijkstra—that communicate with each across levels in the hierarchy and thus achieve pipeline concurrency to implement high-performance real-time system [33].

McCabe's work [32] provides strong theoretical support to our ideas. McCabe argues that tracking the cyclomatic complexity 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. 4 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. 4 has essential complexity (ec) of 1. Likewise, every unstructured CFG with *m* control subgraphs has essential complexity,

$$\mathrm{ec} = \mathrm{C} - \mathrm{m} \tag{1}$$

where C is its cyclomatic complexity. 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 cyclomatic complexity.

$$\mathrm{ec} = \mathrm{C} - 0 = \mathrm{C} \tag{2}$$

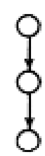

Fig. 4. The CFG with unit essential complexity.

Each removed control graph can be implemented as a separate module. In other words, whether a structured or unstructured graph, the process of modularization involves reducing its cyclomatic complexity to a suitable essential complexity. Composition is a related process. One starts with a CFG of suitable complexity and as more and more nodes are implemented as interface invocations/calls to separately developed modules or components, some of which could be third-party or COTs, the overall complexity of the program increases. Significantly, to compute overall cyclomatic complexity of the program, McCabe presents a result [32]. He provides justification using an example as reproduced in Fig. 5. 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.

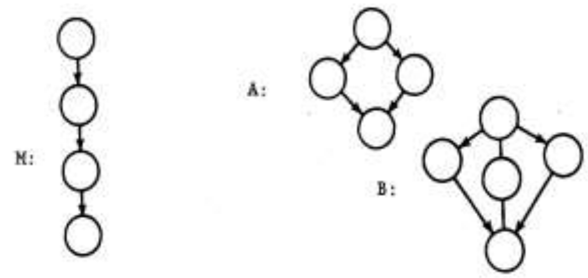

Fig.5. McCabe's example.

The reason is that the main routine maintains its abstract sequential control. 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. 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. If it is an asynchronous call, the main routine does not even suspend. For example, in JAX-RS, an asynchronous http method invocation is set up as a computation node of the type CompletionStage<T>, where T is the return type of the method [9]. The call to an http method returns the CompletionStage<T> instance immediately after spawning a thread (non-request) to carry out the actual computation. At a later stage, the thread calls this instance to complete the computation. It does not disturb the ongoing control flow of the program that spawned it.

Applying the formula for connected components to the example in Fig.5 with p=3, the complexity C is,

$$\mathrm{C} = \mathrm{e} - \mathrm{n} + 2\mathrm{p} = 13 - 13 + 2\mathrm{x}3 = 6 \qquad (3)$$

Also,

$$\mathrm{C} = \mathrm{C(M)} + \mathrm{C(A)} + \mathrm{C(B)} = 2 + 2 + 2 = 6 \qquad (4)$$

In general, the complexity of a collection of k control graphs is equal to the summation of their individual complexities,

$$C(G) = e - n + 2p = \sum_1^k e_i - \sum_1^k n_i + 2k = \sum_i^k (e_i + n_i + 2) = \sum_i^k C_i \qquad (5)$$

McCabe's work signifies that if a large application system (such as a service in our context) is broken up into a main program and subroutines, clearly specifying the logical boundaries of such individual components can help us compute aggregate/overall properties.

4. An Improved Formal Model of a Service-Oriented System

We found no model, other than the Perepletchikov-Ryan-Frampton-Schmidt model [40]-[43], which follows a bottom-up approach and explicitly attempts to define of the logical boundary of a service. Moreover, the model extends the widely-cited generic graph-theoretic model for a software application system by Briand et al. [7].

First, we summarize the Perepletchikov-Ryan-Frampton-Schmidt model. 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, *e*. Given a system, *SYS*, a service *s* can be defined as:

$s = \langle si_s, BPS_s, C_s, I_s, P_s, H_s, R_s \rangle$ 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-module) defined on EXE, i.e., $R \subseteq EXE$, where E is the set of all service implementation elements *e's*, i.e. E= SI ⋃BPS⋃C⋃I ⋃P⋃ H. R is the set of all common and possible relationships of the system SOS. The static coupling relationships of service *s*, R_s, can be categorized as:

Interface to implementation relationships, IIR(s) = $\{(si, e): si = si_s \wedge e \in (BPS_s \cup C_s \cup P_s)\}$ (6)

Internal service relationships, ISR(*s*) = $\{(e_1, e_2): e_1, e_2 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)$ (7)

Incoming relationships, IR(s)

$= \{(e_1, e_2): e_1 \in (BPS\text{-}BPS_s \cup C\text{-}C_s \cup I - I_s \cup P - P_s \cup H - H_s) \wedge e_2 \in (BPS_s \cup C_s \cup I_s \cup P_s \cup H_s)\}$ (8)

Outgoing relationships, OR(s)

$= \{(e_1, e_2): e_1 \in (BPS_S \cup C_s \cup I_s \cup P_s \cup H_s) \wedge e_2 \in (BPS\text{-}BPS_s \cup C\text{-} C_s \cup I - I_s \cup P - P_s \cup H - H_s)\}$ (9)

Service incoming relationships, SIR(s) = $\{(e, si): e \in (BPS - BPS_s \cup C - C_S \cup P - P_s) \wedge si= si_s\}$ (10)

Service outgoing relationships, SOR(s) = $\{(e, si): e \in (BPS_s \cup C_s \cup P_s) \wedge si \neq si_s\}$ (11)

R_s=IIR(s)⋃ISR(s)⋃IR(s)⋃OR(s) ⋃SIR(s) ⋃SOR(s) (12)

In general, any static model tries to estimate what will happen at the later stages of lifecycle [10]. For example, some static dependencies are resolved at run-time. Header-file dependencies are resolved at compile time. However, some concerns 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 CSes, 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) ⋃OR(s) = Φ) but includes them for composite services (SOR(s)⋃OR(s) ≠ Φ). 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, a CDI-style bean that is defined as a JAX-RS root resource class [9] as in the Listing 1 would be exposed as an atomic service. The element type e_1, the root resource class, shows dependency on another element type e_2, a container-managed component, *MyOtherCdiBean*. The element e_2 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 [4, 14]. 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 [23]. It is clear that 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, from the kind of composite services possible to implement using standard application programming frameworks (e.g. Java EE). Hansen [22] calls such applications "enterprise-quality SOA applications."

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 [23] specifies that any service, whether atomic or composite, would itself be an element of SOS.

The above points 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. 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. The logical boundary can be defined similarly for elements other than services as well. However for elements like header files (elements of H; never instantiated) or OO interfaces (element of I; do not have any execution), no such special definition is required. For example, for a header-file, the source file itself serves as the logical boundary. All other header files embedded by include-relationship are elements of its outgoing coupling. If a header-file is being *reused* across elements (by *include*), each such reuse is an incoming coupling of that file.

III. An SOS should be defined as SO*S* =<SI, CPS, C, I, P, H, A, 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.

Regarding the points I) and II) above, as we explained in the Section 3, for example, the underlying context-switches in the case of a uniprocessor machine only signifies sequential machine control transfer and not the transfer of the abstract sequential control of a CFG.

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

Listing 1. A JAX-RS root resource class.

Even in the case of threads, for example, in Java, calls isAlive() and join() that a thread might make on another thread does not branch the individual sequential control flow of either thread [11][50]. In the event the threads are communicating amongst themselves using wait(), notify() or notifyAll() while sharing a synchronized object, the threads do not branch out the sequential control flow of any thread or make a unique control entry into any thread. A call wait() by a thread causes it to stop and a notify() or notifyAll() by another thread is a message to the waiting thread(s) to resume. As soon as a waiting thread receives a message from notify() or notifyAll(), the call wait() can treated to be over.

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 $si_s \in SI \wedge \{(C_s \subseteq C \wedge I_s \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H) \wedge C_s \cup I_s \cup P_s \cup H_s = D(f_s) \wedge (Rs \subseteq R)\}$. f_s, the logical boundary the service *s*. Only elements that are 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 regarded as exclusively belonging 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=$<si_s, CPS_s, C_s, I_s, P_s, H_s, A_s, f_s, R_s>$ is also a service of *SYS* if and only if $si_s \in SI \wedge \{(CPS_s \subseteq CPS \wedge C_s \subseteq C \wedge I \subseteq I \wedge P_s \subseteq P \wedge H_s \subseteq H \wedge A_s \subseteq A) \wedge C_s \cup I_s \cup P_s \cup H_s = D(f_s) \wedge (Rs \subseteq R)\}$. Such a service is called a composite service.
$R \subseteq E \times E$, where E is the set of all elements (modules), *e'*s, i.e. E= SI⋃C⋃I⋃P⋃H⋃A⋃CPS. R is the set of all common and possible relationships of an SOS.

With the definition of logical boundary of a service as above, statically-resolved dependencies like global function calls in C++ or static function calls in Java would be removed from the set ISR(s) and would be typed along with injected dependencies by the elements of set OR(s). SOR(s) gets merged with OR(s). Services share most components and resources of the system except those within their respective sets $D(f_s)$. However, for example, if a C++ header file containing inline functions is reused in different elements or services, it is not typed by an element belonging to the set $D(f_s)$. It will be an element of OR(s). All such dependencies (that are not calls/invocations to functions, injected dependencies etc.) will be typed by elements included in the set $DD(f_s)$. The set IR(s) and SIR(s) are merged as IR(s). All incoming coupling is typed by a service interface. Nodes typed by $D(f_s)$ are not directly coupled to any element outside of the service. This notion of a service is similar to one for service composite in the SCA paradigm of SOA. Thus, for an atomic service, *s:*

$$OR(s) = E(f_s) \cup DD(f_s) \quad (13)$$

E() extracts the *set* of unique elements, *e*'s, corresponding to various dependency invocations/calls, i.e., invocations whose executions are not inlined.

$$IR(s) = \{(e, si): e \in (C - C_s \cup P - P_s) \wedge si = si_s\} \quad (14)$$

$$C_s,\ P_s \in D(f_s)$$

$$C_s \cup I_s \cup P_s \cup H_s = D(f_s) \quad (15)$$

(13)

For a composite service, the only change is in IR(s),

$$IR(s)=\{(e, si): e \in (CPS-\{s\} \cup C-C_s \cup P-P_s \cup A-\{s\}) \wedge si=si_s\} \quad (16)$$

$$C_s,\ P_s \in D(f_s)$$

We can specialize this model to the microservices architecture (MSA) style. MSA is a subset of the SOA style [3]. A microservice is a highly autonomous software component that cannot be composed out of other microservices or services. A microservice is characterized by inter-related characteristics of service independence, single responsibility, self-containment, high decoupling, high resilience and decentralized data management. The applications built using MSA should keep the microservices decoupled and fully independent. Any choreography in an MSA is performed by the initiating application and not from within or by the microservices. Thus, an atomic service *as* of an SOS is a microservice if the responsibility of development and maintenance of *as* as also of *most* of the dependencies participating in OR(*as*) lies with a single team .

We have discussed theoretical foundation for our logical-boundary definition in the Section 3. Here we discuss some more supporting ideas. The original model associates a set of classes, OO interfaces, package headers etc. to one particular service interface element as its logical boundary. Apart from the concerns mentioned earlier in this section, it is in conflict with reuse of such elements across services. For example, package headers, in any case, are required to even inline functions defined within them; they could be reused across services. Services exposed out of legacy systems might be reusing a lot of elements across services. Moreover, the original model excludes from the logical boundary of a service the ownership of programming logic/algorithms (also implementable in an elements in D()) that could be unique to the service. The definition of logical boundary should be technology-agnostic (e.g., unlike logical grouping package in Java or namespace in XML) and physical-packaging-independent (ultimately packages and namespaces are resolved from specific files). A definition of logical boundary that can be resolved with respect to a universal convention is what we are seeking. For example, a layer in TCP/IP stack serves as boundary for calls from the layers adjacent to it. Developers implement it in operating systems and both users and developers can delineate this boundary with respect to the universal standard TCP/IP protocol stack they follow. CFG is also a universal convention. Our definition of logical boundary in terms of CFG addresses all these concerns as well.

CFG is an important tool for analyzing structured and object-oriented programs [5][10][15]. A program's CFG is a necessity to calculate its cyclomatic complexity. Cyclomatic complexity provides upper bound on the number of test cases that will be required to ensure that every statement in the program is executed at least once [45]. Ito [24] shows that, for another important type of graphs, program dependence graphs (PDGs), employed in static analysis by a compiler, PDGs constructed for usual programs are deterministic and that such PDGs are semantically equivalent to the corresponding CFGs.

If services are being developed afresh, due care can be taken during design-time to ensure that CFGs are available early-on. Methods to extract control flow graphs from UML sequence diagrams are described in [15][28]. On the other hand, if services are being exposed from legacy code and components, there are several static analysis tools that can help in generating CFGs.

Amighi et al. and Gomes et al. [2][3][16] report techniques to extract incremental, modular CFGs from incomplete Java bytecode programs with exceptions. They argue that such techniques would be handy in the event that some components are not available for systems under development. If at all such components become available, there source code might not be available, for example, in the case of third-party software. Diniz and Diogo [12] report automatic extraction of CFGs by process mining. Kirkegaard and Moller [26] describe a tool for generating, at compile time, sound CFGs from web applications constructed with Java servlets and JSP scripts. Halfond [21] describes tool for generating CGFs from web applications. Jovanovic et al. [25] describe a tool that converts each PHP script file of a web application that is visible in a browser into CFG as an intermediate result. Yang et al. [54] describe a tool that generates CFGs from web applications. In [34], Monga et al. report a tool that converts a web application constructed form PHP scripts into a CFG. These tools are applicable to web services since a web service is basically a web application with a service interface (API) in lieu of a user-interface/frontend.

The significance of CFGs and availability of tools to automatically extract them support our choice of CFG as a formal construct to represent the logical boundary of a service.

5. Metrics

Basic metrics are readily available from the model. The metric, incoming coupling of service, *ic(s)*, is

$$ic(s) = |IR(s)| \tag{17}$$

The metric, outgoing coupling of a service, *oc(s*), is

$$oc(s) = |OR(s)| \tag{18}$$

For an atomic service *s*, let the logical boundary an operation of an atomic service *s* be f_o.
Collect all elements D(f_o), DD(f_o) and E(f_o) into a set. Count *common* elements from such sets across all the operations *oi* of the service *s*. Let this be denoted by *count1*.

$$count1 = |\cap_i[D(f_{oi}) \cup DD(f_{oi}) \cup E(f_{oi})]| \tag{19}$$

Count total unique elements across all the sets. Let this be denoted by *count2*.

$$count2 = |\cup_i[D(f_{oi}) \cup DD(f_{oi}) \cup E(f_{oi})]| \tag{20}$$

The cohesion of the service *s*, *coh(s)*, is

$$\begin{aligned} &If\ count1 = 0, \quad coh(s) = 0 \\ &Otherwise, \quad coh(s) = \frac{count1}{count2} \end{aligned} \tag{21}$$

If very low, due consideration should be given to split operations as separate atomic services.

If a service *s* has no outgoing coupling OR(s), we consider it to have lowest instability. We assume this value as 1. Suppose it has outgoing coupling |OR(s)|=*m*. We model absolute instability of *s* as follows

$$ins(s) = 1 + \sum_{1}^{m} w_i \tag{22}$$

w_i is the weight (a positive integer) assigned to the i^{th} element of OR(s) in proportion to the degradation it may cause to the overall functionality of *s* in the event of being unavailable due to maintenance, breakdown etc. For example, if a service has 5 public operations. If i^{th} element of OR(s) degrades any two public operations, w_i =2.

Degree of self-containment of a service *s*, *sc(s)*, reflects its stability, that is, the extent to which it does not depend on outgoing coupling. It also signifies the extent to which it would be coupled more through its service interface (incoming coupling) and thus be more loosely coupled.

$$sc(s) = \frac{1}{ins(s)} \tag{23}$$

We consider |IR(s)| to be the absolute criticality of the service *s* [6]. We define relative criticality of the service *s*, *rcr(s)*, as,

$$rcr(s) = |IR(s)| * ins(s) \tag{24}$$

Suppose two services s_1 and s_2 have equally high absolute criticalities $|IR(s_1)|$ and $|IR(s_2)|$ respectively. If s_1's absolute instability $ins(s_1)$ is higher than s_2's absolute instability $ins(s_2)$, s_1 is at more risk of getting unavailable and thus requires *relatively* more critical attention than s_2.

6. Conclusion and Future Work

We described the concept of logical boundary of a service in concrete terms. An improved and comprehensive formal model of service-oriented systems was presented and its utility in defining some novel design metrics was shown. It was explained that the model can explain a microservice too. We discussed many existing theoretical and practical concepts from computer science and software engineering to ground our ideas. Our ideas are also broadly applicable to large, distributed and component-based software systems. In future work, we intend to take forward the work reported here and elaborate using many diverse application software scenarios and domains.

REFERENCES

[1] Amighi, A, & Gomes, P., Gurov, D. & Huisman, M., Sound Control-Flow Graph Extraction for Java Programs with Exceptions, 33-47, 2012 https://doi.org/10.1007/978-3-642-33826-7_3

[2] Amighi, A., Gomes, P., Gurov, D. *et al*. Provably correct control flow graphs from Java bytecode programs with exceptions, *Int J Softw Tools Technol Transfer* 18, 653–684, 2016 https://doi.org/10.1007/s10009-015-0375-0

[3] Balakrushnan, S. et al., Microservices Architecture, The Open Group, San Francisco. CA, USA, 2016 https://publications.opengroup.org/w169

[4] Barry, D.K. & Dick, D., *Architectures, and Cloud Computing:* The Savvy Manager's Guide, Second Edition, Elsevier Inc., 2013

[5] Besson, F. & Jensen, T. & Métayer, D., Model Checking Security Properties of Control Flow Graphs, *Journal of Computer Security*, 9. 217-250. 10.3233/JCS-2001-9303, 2001

[6] Bishop P., Bloomfield R., Clement T. & Guerra S.,Software Criticality Analysis of COTS/SOUP, In: Anderson S., Felici M., Bologna S. (eds) *Computer Safety, Reliability and Security*. SAFECOMP 2002, Lecture Notes in Computer Science, vol 243, Springer, Berlin, Heidelberg, 2002

[7] Briand, L.C., Morasca, S. and Basili, V.R., Property-based software engineering measurement, In *IEEE Transactions on Software Engineering*, vol. 22, no. 1, pp. 68-86, 1996, doi: 10.1109/32.481535

[8] Broy, M., A Theory of System Interaction: Components, Interfaces, and Services, In: Dina, G., Scott, A.S. & Wegner, P. (Eds.) *Interactive Computation: The New paradigm*, Springer, 2006

[9] Bucek, P. & Pericas-Geertsen, S. (eds.), *JAX-RS: Java™ API for RESTful Web Services*, Version 2.1, Final Release, Oracle Corp., US, 2017

[10] Cooper, K.D. and Torczon, L., *Engineering a Compiler*, 2nd Ed., Elsevier Inc., US, 2012

[11] Dijkstra, E.W., The Structure of the "THE"-Multiprogramming System, *Comms. of the ACM*, vol. 11, No. 5, 1968

[12] Diniz, P.C. & Diogo R. F., Automatic Extraction of Process Control Flow from I/O Operations, In: *6th International Conference on Business Process Management (BPM 2008)*, volume 5240 of Lecture Notes in Computer Science, Springer, 2008

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

[14] Ganci, J., *Patterns: SOA Foundation Service Creation Scenario*, 2006 http://www.redbooks.ibm.com/redbooks/pdfs/sg247240.pdf

[15] Garousi, V. *et al.*, Control Flow Analysis of UML 2.0 Sequence Diagrams, Carleton University TR SCE-05-09, 2005

[16] Gomes, P. & Picoco, A. & Gurov, D., Sound Control Flow Graph Extraction from Incomplete Java Bytecode Programs, 2014 10.1007/978-3-642-54804-8_15.

[17] Gonen, B. et al., Maintaining SOA Systems of the Future: *How Can Ontological Modeling Help,* In Proceedings of the International Conference on Knowledge Engineering and Ontology Development (KEOD-2014), pages376-381, ISBN: 978-989-758-049-9, 2014

[18] Gruhn, V. & Laue, R., Complexity Metrics for Business Process Models, http://ebus.informatik.uni-leipzig.de/~laue/papers/metriken.pdf

[19] Guidi, C. & Lucchi, R., Formalizing mobility in Service Oriented Computing, *Journal of Software*, vol. 2, no. 1, 2007

[20] Habermann, A.N., Lawrence, F. & Cooprider, L., Modularization and Hierarchy in a Family of Operating Systems, *Comm. ACM*, vol. 19, No. 5, 1976

[21] Halfond W.G.J., Identifying Inter-Component Control Flow in Web Applications. In: Cimiano P., Frasincar F., Houben GJ., Schwabe D. (eds) *Engineering the Web in the Big Data Era*. ICWE 2015, Lecture Notes in Computer Science, vol 9114. Springer, Cham, 2015

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

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

[24] Ito, S., Semantical Equivalence of the Control Flow Graph and the Program Dependence Graph, arXiv:1803.02976v1 [cs.PL], 2018

[25] Jovanovic, N. et al., Precise Alias Analysis for Static Detection of Web Application Vulnerabilities, *PLAS'06*, Ottawa, Ontario, Canada. ACM 1-59593-374-3/06/0006, 2006

[26] Kirkegaard, C. & Moller, A., Static Analysis for Java Servlets and JSP, *BRICS Report Series* , RS-06-10, ISSN 0909-0878, Department of Computer Science, University of Aarhus, Denmark, 2006 http://www.brics.dk/RS/06/10/

[27] Korostelev, A. *et al*., Error Detection in Service-Oriented Distributed Systems, *Proc. of IEEE Int. Conf. on DSN 2006*, vol. 2, pp. 278-282, Philadelphia, USA, 2006

[28] Kundu, D., Samanta, D. and Mall, R., An Approach to Convert XMI Representation of UML 2.x Interaction Diagram into Control Flow Graph, *ISRN Software Engineering* Volume 2012, Article ID 265235, 2012 doi:10.5402/2012/265235

[29] Liu, Y. & Traore, I., Complexity Measures for Secure service-Oriented Software Architectures, *In the Proc. of the 3rd IEEE International PROMISE Workshop*, Minneapolis, Minnesota, USA, 2007

[30] Mao, C., Control Flow Complexity Metrics for Petri Net based Web Service Composition, *Journal of Software*, Vol. 5, No. 11, 2010

[31] Massuthe, P., Wolfgang, R. and Schmidt, K., An Operating Guideline Approach to the SOA, *Annals of Mathematics, Computing & Teleinformatics*, VOL 1, NO 3, 2005

[32] McCabe, T.J., A Complexity Measure, *IEEE Transactions on Software Engineering*, Vol. Se-2, No. 4, 1976

[33] Michaloski, J.L., Wheatley, T.E. and Lumia, R., Analysis of Computational Parallelism with a Concurrent Hierarchical Robot Control System, NISTIR 90-4251, NIST, US, 1990

[34] Monga, M. & Paleari, R. & Passerini, E., A hybrid analysis framework for detecting web application vulnerabilities, 25-32, 10.1109/IWSESS.2009.5068455, 2009

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

[36] Parnas, D.L., On a "Buzzword": Hierarchical Structure. In: Gries D. (eds) *Programming Methodology. Texts and Monographs in Computer Science*. Springer, New York, NY, 1978

[37] Parnas, D.L., Designing Software for Ease of Extension and Contraction, *IEEE Trans. SE*, vol SE-5, No. 2, 1979

[38] Parnas, D.L., On the Criteria To Be Used in Decomposing Systems into Modules, *Comm. ACM*, vol 15, No. 12, 1972

[39] Parnas, D.L., On the Design and Development of Program Families, *IEEE Trans. SE*, vol SE-2, No. 1, 1976

[40] Perepletchikov, M., Ryan , C. and Frampton, K., Cohesion Metrics for Predicting Maintainability of Service-Oriented Software, *In 7th International Conference on Quality Software*, Portland, USA, 2007

[41] Perepletchikov, M., Ryan , C. and Frampton, K., Coupling Metrics for Predicting Maintainability in Service-Oriented Designs, *In 18th International Conference on Software Engineering (ASWEC2007)*, Melbourne, Australia, 2007

[42] Perepletchikov, M., Ryan , C., Frampton, K. and Schmidt, H., Formalising Service-Oriented Design, *Journal of Software*, Vol.3, No. 2, 2008

[43] Perepletchikov, M., *Software Design Metrics for Predicting Maintainabaility of Service-Oriented Software*, Ph.D. Thesis, RMIT Univ., Melbourne, 2009 https://researchbank.rmit.edu.au/view/rmit:1479/n2006006582.pdf

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

[45] Pressman, R. S., *Software Engineering: A Practitioner's Approach* Sixth Ed., Int. Ed., McGraw-Hill, 2005

[46] Ravindran, B., Engineering dynamic real-time distributed systems: architecture, system description language, and middleware, in *IEEE Transactions on Software Engineering*, vol. 28, no. 1, pp. 30-57, 2002, doi: 10.1109/32.979988.

[47] Rud, D., Schmietendorf , A., and Dumke, R., Resource metrics for service-oriented infrastructures, *In Proc. SEMSOA 2007*, pp. 90-98, May 10-11, Hannover, Germany, 2007 http://www.cs.uni-magdeburg.de/~rud/papers/Rud-13.pdf

[48] Rud, D., Schmietendorf, A., and Dumke, R. Dumke, Product metrics for service-oriented infrastructures, *In Proc. 16th International Workshop on Software Measurement/DASMA Metrik Kongress 2006*, Potsdam, Germany, 2006 http://www.cs.uni-magdeburg.de/~rud/papers/Rud-07.pdf

[49] Russell, D. and Xu, J., Service Oriented Architecture in the Provision of Military Capability, *UK e-Science All Hands Meeting*, 2007 http://www.comp.leeds.ac.uk/NEC/doc/SOACapabilityAHM2007.pdf

[50] Schildt, H., *The Complete Reference: Java*, 9th Ed., McGraw Hill Education, 2014

[51] *Service Component Architecture: Assembly Model Specification*, SCA Version 1.00, 2007 https://web.archive.org/web/20070712102723/http://www.osoa.org/display/Main/Service+Component+Architecture+Specifications

[52] Silberschatz, A., Galvin, P.B. and Gagne, G., *Operating System Concepts*, 8th Ed., John Wiley & Sons, US, 2009

[53] Xu, T., Qian, K. and He, X., Service Oriented Dynamic Decoupling Metrics, *The 2006 Intl. Conf. on Semantic Web and Web Services (SWWS' 06), June 26-29, 2006 WORLDCOMP' 06*, Las Vegas, USA, 2006

[54] Yang, J-T et al., Constructing an Object-Oriented Architecture for Web Application Testing, *Journal of Information Science and Engineering 18*, 59-84, 2002

[55] Zhao, W., Liu, Y., J. Zhu & Su, H., Towards Facilitating Development of SOA Application with Design Metrics, *Service-Oriented Computing - ICSOC 2006, 4th International Conference*, Chicago, IL, USA, 2006