IEEE Access
Multidisciplinary ⋮ Rapid Review ⋮ Open Access Journal

Received June 23, 2020, accepted July 7, 2020, date of publication July 13, 2020, date of current version July 23, 2020.

Digital Object Identifier 10.1109/ACCESS.2020.3008904

Static Analysis for Improved Modularity of Procedural Web Application Programming Interfaces

ALISTAIR BARROS, CHUN OUYANG, AND FUGUO WEI
School of Information Systems, Queensland University of Technology, Brisbane, QLD 4000, Australia

Corresponding author: Chun Ouyang (c.ouyang@qut.edu.au)

The work of Alistair Barros and Chun Ouyang was supported by the Australian Research Council (ARC) Discovery Grant DP190100314.

ABSTRACT Despite their rapid growth, the utilisation of application programming interfaces (APIs) poses challenges for companies under pressure to yield productive systems integration. APIs of larger systems tend to be large, complex and have reduced modularity and quality, which makes them cumbersome to comprehend and use. These challenges can be addressed by static API analysis that focuses on studying API code itself and deriving business entities and dependencies from operational signatures. However, existing techniques for static analysis of APIs face the challenges in deriving a sufficient coverage of business entity relationship types from implementation-oriented API operational signatures carrying limited semantic insights. The paper aims to address such problems by supporting static analysis techniques for APIs that improve their modularity. Our approach adopts an object-oriented paradigm where the concept of ''object'' is exemplified by the notion of business entity. It systematically applies interface analysis methods and techniques for eliciting knowledge of business entities and their attributes, for deriving the temporal order of calling operations across multiple business entities, and for learning and extracting various ways of invoking a service via APIs. The approach is implemented as an open-source tool and applied to a group of widely-deployed services in practice for validation. The research contributes to identifying key aspects of both the structure and behaviour of APIs, which will lead to building a simplified but comprehensive interface (presentation) layer to assist service users in understanding complex and overloaded interfaces as well as to facilitate efficient and effective service integration.

INDEX TERMS Application programming interfaces, APIs, business entity, service interface analysis, service interface synthesis, service variants, web services.

I. INTRODUCTION

Web-based application programming interfaces (APIs) are critical for enabling organisations to open up software applications through partner ecosystems and the Internet. APIs provide operational signatures to create, read, update and delete data, related to business (or logical) entities, managed in software applications, without revealing the software implementation that supports the operations. Their descriptions captured through widely supported interface description languages in Web Services Definition Language (WSDL) and Representational State Transfer (REST) API combined with the encapsulated nature of operations they expose, promote flexible ways of accessing and composing software applications. The significance of APIs can be seen through their support in Internet platforms, e.g. Facebook, Twitter, YouTube, Amazon, eBay and Google Cloud Platform, as well as enterprise systems prevalent in corporate sectors, e.g. SAP Business Hub and Oracle Peoplesoft APIs. Moreover, central API repositories, such as Programmable Web with around 19,000 APIs, are fuelling strategic interest in APIs as evident by the notion of the API Economy [1].

Despite their rapid growth, the utilisation of APIs poses challenges for companies under pressure to yield productive systems integration [2]. Their specifications are essentially technical and the user documentation is targeted at programmers, if available at all [3]. APIs of larger systems, especially, tend to be large, complex and have reduced modularity and quality, given many maintenance cycles of systems. This makes them cumbersome to comprehend and

The associate editor coordinating the review of this manuscript and approving it for publication was Zhangbing Zhou.

This work is licensed under a Creative Commons Attribution 4.0 License. For more information, see https://creativecommons.org/licenses/by/4.0/

use. As examples, WSDL-based APIs of SAP and Oracle enterprise resource planning (ERP) systems have up to three hundred parameters per operation and multiple levels of nesting while FedEx's shipment service API has around 1000 parameters and 9 levels of nesting [4].

Techniques have been proposed to automatically analyse APIs, reverse-engineer structural and behavioural properties, and make recommendations for improvement. Many techniques have focused on dynamic analysis, which concerns the mining of executed API interactions recorded in systems logs (such as send/receive interactions on operations and the data payload). The service deployment data recorded in systems logs can also be used to analyse non-functional requirements of APIs. One aspect of dynamic API analysis is deriving message formats of APIs [5], given that API documentation may be missing, incomplete or inaccurate regarding specific usage details. Another aspect is deriving message correlations in service interactions (e.g., a set of sales orders are related to a delivery order and a set of delivery orders are related to an invoice). In [6], message correlations are based on casually related request-response interactions, through which different messages are passed (e.g., sales order and delivery order). Other aspects involve discovery of service interaction processes [7], [8] based on operation sequences involving correlated messages, i.e., discovery of service orchestration/choreography. Finally, the derivation of business entities inherent in message types of API services and their dependence can be inferred via analysis of log data [9].

More recently, techniques have been developed for static analysis of APIs. The static API analysis techniques target procedural APIs and WSDL specifically. This is because this style of API is the oldest and most difficult to comprehend, use, and improve for quality. Procedural APIs have parameters which are based on simple and complex (nested) data types, and lack data structure abstractions. As such, they pose the greatest challenge for static API analysis techniques, especially for the large sized operational signatures (in the realm of hundreds of parameters). Existing techniques focus on analysing API code itself and deriving business entities and dependencies from operational signatures. Specifically, they use heuristics for parameter cohesion [10] to identify business entities, entity co-location in operations, and operation input and output dependencies to derive business entity relationships [11]. This knowledge can be used to assess problems of operation overloading and to make recommendations for improved modularity, where operations concern individual entities only. This is beneficial not only for improving flexible API access and composition, but also for evolving older, procedure-oriented WSDL APIs into more contemporary document-oriented WSDL or REST APIs. Extracted structural and behavioural properties from code can further improve the productivity of semantic tagging of APIs based on ontologies, to improve their search and access. In addition, static API analysis can help contextualise dynamic analysis of execution data, e.g., for improved insights into operational dependencies, based on business entity relationships, and data exchange analysis. To date, the challenge of static API analysis can be summarised as deriving a sufficient coverage of business entity relationship types from implementation-oriented API operational signatures carrying limited semantic insights.

This paper provides a state-of-the-art exposition of static API analysis. It aims to address the complexity of APIs by proposing a systematic approach that supports static analysis of API specifications for improved modularity. The approach builds on the notion of business entity and systematically applies interface analysis methods and techniques for eliciting knowledge of business entities and their attributes, for deriving the temporal order of calling operations across multiple business entities, and for learning and extracting various ways of invoking a service via APIs. It is implemented and applied to a group of widely-deployed services in practice for validation. The research contributes to identifying key aspects of both the structure and behaviour of APIs, which will lead to building a simplified but comprehensive interface (presentation) layer to assist service users in understanding complex and overloaded interfaces as well as to facilitate efficient and effective service integration.

The research presented in this paper draws upon and extending the techniques developed from our previous contributions [12]–[14]. An earlier version of our method for extracting business entities from a service interface specification was proposed in [12], and our initial findings in deriving the potential temporal order of operations that may be carried out by different business entities was reported in [13]. In [14], we presented a service variant analysis technique which can be used to compute different ways of invoking a service based on the service's business entity model, and no further extension is made to this technique in our current study. However, given that this technique is an important part of the overall systematic approach that we propose for static analysis of Web APIs, a short introduction to the technique published in [14] is included in the paper. Furthermore, having a complete proposal of our systematic approach allows us to present an overall tool implementation of the approach and its application to several services that are widely deployed in practice.

It is also worth noting that our approach focuses on using WSDL as a specific procedural API language. The choice of WSDL is twofold. Firstly, it is the most prominent API description language and was designed as an independent (API) definition language inspired by the CORBA IDL specification. It maps into different languages such as Java, RFC API, PHP (SoapClient). Importantly, previous static API analysis techniques focused on procedural APIs and WSDL specifically (e.g., [15]–[19]). Another reason that our techniques have not supported REST API is that it is not a procedural language, as the data types are reflected through resources. As such, REST API fosters modular design of APIs. There are other issues that can arise through REST APIs, such as consistency of resource structure and

decomposition addressed through research into REST API anti-patterns by Palma *et al.* [20]. However, the problem that we have specifically addressed is extracting entities from overloaded API operation signatures with parameters that relate to multiple entities or multiple variants of the same entity. Such over-loading is less of a concern for REST APIs given that HTTP CRUD operations (e.g., POST, GET, PUT, DELETE) manipulate data through resource abstractions.

The rest of the paper is structured as follows. Section II elaborates the research motivation via an exemplar scenario and states the research problem. Section III presents our approach consisting of three key building blocks, namely structural interface analysis, behavioural interface synthesis, and service variants analysis. Section IV discusses evaluation of the approach via the implementation of a prototypical tool and the experiment results of applying the tool to a group of widely-deployed services in practice. Section V reviews existing studies on API analysis. Section VI concludes the paper and outlines the future work.

II. PROBLEM STATEMENT

Let us start with a motivating example about a shipping service. A manufacturing company called Smith Brothers (a fictional name) wished to incorporate a shipment service into its web service enabled systems so that it can ship goods and manage shipping orders through its systems. The company identified a number of shipment service providers such as FedEx, UPS and DHL, which all offered web service interfaces to their users, but these interfaces were very complex. For instance, the FedEx Open Shipping service API specification written in WSDL has 7727 lines and more than 1000 parameters[1] (see Listing 1 for a fraction of this specification). Many of these parameters are of a complex data type and hierarchically structured. What FedEx has provided is just a 645-page pdf document,[2] which depicts the details of what each parameter means for programmers.

The APIs of enterprise services, such as those from SAP, FedEx, and Amazon, are often *complex and overloaded due to inherited complexity from legacy systems*. Most service providers, especially enterprise system vendors, simply migrate their legacy systems to services with heavy operational signatures wrapped in WSDL specifications [21]. The aforementioned FedEx Open Shipping service specification is a typical example. These XML-based documents usually contain thousands of lines of codes that attempt to describe the input and output messages of each operation offered by a service. For example, the aforementioned FedEx Open Shipping service specifies thousands of parameters.

Often the APIs of enterprise services are a result of a direct migration from legacy systems that have a large number of input parameters catering for various needs and different

```
<definitions targetNamespace="http://fedex.com/ws/
    openship/v9" name="OpenShipServiceDefinitions">
  <types>
      <xs:complexType name="RequestedShipment">
        <xs:sequence>
          <xs:element name="ShipTimestamp" type="
              xs:dateTime" minOccurs="0"/>
          <xs:element name="Shipper" type="ns:Party"
              minOccurs="0"/>
          <xs:element name="Recipient" type="ns:Party"
              minOccurs="0"/>
          <xs:element name="ShippingChargesPayment" type=
              "ns:Payment" minOccurs="0"/>
          <xs:element name="RequestedPackageLineItems"
              type="ns:RequestedPackageLineItem"
              minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
      </xs:complexType>
      <xs:complexType name="RequestedPackageLineItem">
        <xs:sequence>
          <xs:element name="SequenceNumber" type="
              xs:positiveInteger" minOccurs="0"/>
          <xs:element name="TrackingIds" type="
              ns:TrackingId" minOccurs="0" maxOccurs="
              unbounded"/>
          <xs:element name="ItemDescription" type="
              xs:string" minOccurs="0"/>
        </xs:sequence>
      </xs:complexType>
  </types>
  .
  .
  .
  <portType name="OpenShipPortType">
    <operation name="createOpenShipment" parameterOrder="
        CreateOpenShipmentRequest">
      <input message="ns:RequestedShipment"/>
      <output message="ns:CreateOpenShipmentReply"/>
    </operation>
    <operation name="addPackagesToOpenShipment"
        parameterOrder="AddPackagesToOpenShipmentRequest"
        >
      <input message="ns:AddPackagesToOpenShipmentRequest
          "/>
      <output message="ns:AddPackagesToOpenShipmentReply"
          />
    </operation>
  </portType>
</definitions>
```

Listing. 1. An excerpt of FedEx Open Shipping service's `CreateOpenShipmentRequest` **WSDL specification .**

users. Significant examples of available Web services exist that have complex and overloaded operational signatures. In addition to the above FedEx's web services for shipment planning, SAP, the largest ERP vendor, has a repository of Web services which demonstrates this. In the SAP R/3 ERP system, the number of fields of LineItems for a Purchase Order is numerous and the header of Purchase Order has hundreds more fields.[3] Also, the Procure-to-Pay service bundle describes WSDL service for creating purchase orders. The WSDL has operations with around 300 parameters and five levels of nesting. The document describes several business entities in the different operations of that service. Another widely set of services are from Amazon which reflect similar operation sizes, nesting and existent of business entities. These legacy ERP fields discussed in [22] have been directly turned into input parameters of web services that are related to the creation of Purchase Order. This approach, referred

[1] https://github.com/jzempel/fedex/blob/master/fedex/wsdls/OpenShipService_v9.wsdl

[2] https://www.fedex.com/templates/components/apps/wpor/secure/downloads/pdf/201607/FedEx_WebServices_OpenShipping_WSDLGuide_v2016.pdf

[3] http://www.stechno.net/sap-tables.html?view=saptable&id=EKKO

to as a "super-service approach" in [5], is a single instance that provides all service capabilities required by all users, while at the same time it yields many different variants – multiple ways of invoking a service. Furthermore, empirical research has shown that direct migration approaches result in low quality service interfaces due to the fact that new systems components often reuse existing systems components with automatically generated WSDLs, which are hard to comprehend by developers [23].

Despite the complexity of APIs, there *lacks a guidance to service users about the service capabilities and valid ways of invoking services*. Hence, service users often find it hard to understand what the services offer and how to invoke these services. In reality, consuming and integrating enterprise services usually requires manual effort and reliance on service providers or domain specialists to provide insights into their APIs [24]. As a result, service integration incurs significant lead times and costly maintenance, and its productivity in the context of dynamic service growth on the scale of the Internet is restricted.

III. APPROACH

This research aims at addressing the complexity of APIs via static analysis. Our approach adopts an object-oriented paradigm where "objects" are exemplified by business artefacts. This section presents the approach.

A. RATIONALE

A traditional WSDL API specification (or API specification for short) defines operations and their input and output parameters using XML codes. In fact, each operation is associated with one or more business artefacts, but these are not specified in the XML codes of WSDL API specifications. Considering the example of FedEx shipment service in Section II, the "createOpenShipment" operation is associated with the business artefact "ShippingOrder" which is not specified in the FedEx Open Shipping service API specification. Business artefacts entail what a service offers in an object-oriented manner. Comparing to hundreds of lines of XML codes, an API specification, if defined using business artefacts associated with operations and their parameters, will be much easier for service users to read thus leading to a better understanding of the service capabilities.

Hence, we introduce the notion of *business entity* to refer to a business artefact mentioned above. More precisely, a business entity is a business-related object being created and having evolved as result of a service invocation. The advantages of applying the idea of business entity are threefold.

Firstly, business entities represent the explicit knowledge concerning a business operational goal [3]. Again, taking the FedEx Open Shipping service for example, "ShippingOrder" is a business entity and to create a shipping order is an operational goal associated with this business entity. Secondly, business entities often are not standalone but relate to each other, and the relations between business entities present useful semantic information. For instance, "ShippingOrder" and "Track" are two correlated business entities that can be derived from the FedEx Shipping service, and this informs the service users that they can track a 'shipping order' using the 'track' operation. Finally, business entities and their relations can be specified using *business entity(-based data) models*. Comparing to XML coding in the existing API specifications, a graphical representation of a business entity model will provide service users with an articulated view of the internal structure of the service and insights into what a service offers, and hence it helps improve the comprehensibility of APIs.

B. OVERVIEW OF THE APPROACH

Our approach builds on the notion of business entity, and as shown in Figure 1, it mainly consists of three stages. The first stage is fundamental, in which an API specification is analysed to extract the business entities and their relations. This stage focuses on *structural interface analysis*, which takes an API specification as an input and yields a *business entity model* as the output. Figure 1 (a) depicts an abstract example illustrating the main idea of this stage. Assume an API specification s contains an operations op_1, which has a set of 13 input parameters at different levels of nesting. The first step is to map each complex parameter of an operation to a business entity and to map each nested parameter of that complex parameter to an attribute of the corresponding business entity. For example, parameter p_1 is mapped to business entity A, and the nested parameters p_2 to p_5 (of p_1) are mapped to the attributes a_0 to a_3 (of A and a_0 is the key attribute). In the second step, the structural relations (e.g. nesting relation) between the complex parameters are used to inform the relations between the derived business entities. For example, the fact that parameter p_5 is nested in p_1 implies that business entity A contains entity B. Following this two-step derivation method, a business entity model representing API specification s can be obtained which comprises four business entities (A to D) and their relations. Such a model presents a simplified and modular representation of the complex API of a service, along with the contextual insights into what the service offers. Design of such a scientific method for structural interface analysis is presented in Section III-C.

In the second stage, our focus proceeds to the derivation of behavioural interface which is concerned with the temporal order of invoking operations across multiple business entities. Given the business entity model extracted from an API specification in the above stage, the business entities and their relations specified in the model can be used to synthesise a behavioural interface for API. Despite the fact that an API often has a large amount of operations, these operations are designed to create, read, update or delete business entities, and thus can be categorised into CRUD operations. Assume that business entity B is contained in (i.e. part of) business entity A and two operation op_1 and op_2 are used to create A and B, respectively. As for behavioural interface, operation op_2 must be invoked before op_1 to ensure that B is created before A can be created. Otherwise, without specifying such order in invoking operations, it may be possible to end up in a

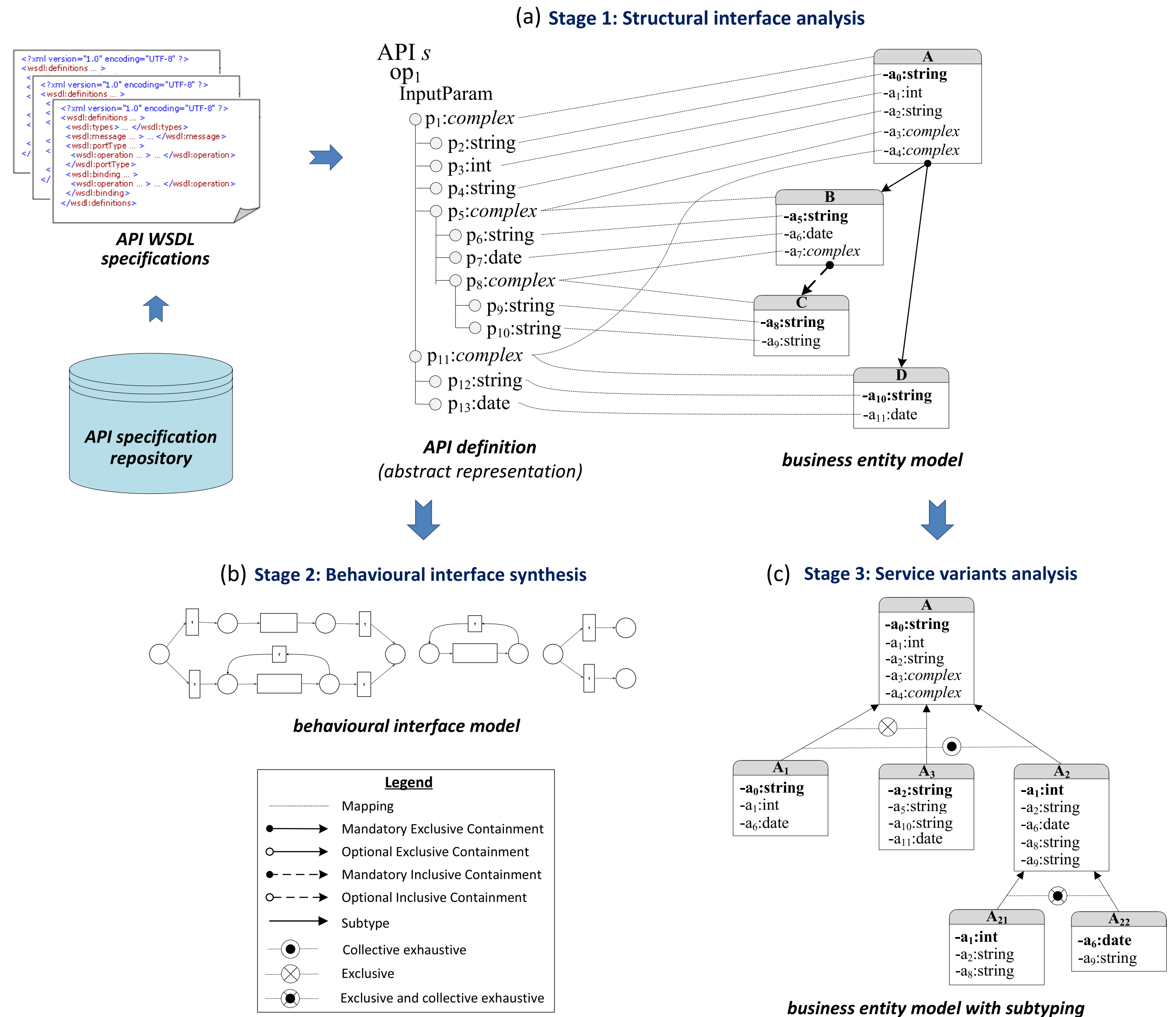

FIGURE 1. Overall approach of static analysis of APIs.

deadlock scenario. For example, if op_2 is to be invoked after op_1, then op_1 can never be completed but waits for entity B to become available (which however will not happen unless op_2 is invoked). Figure 1 (b) depicts an abstract behavioural interface presented in some graphical modelling notation (an abstract representation of a formal modelling language known as Petri nets [25]). Design of behavioural interface synthesis, including behavioural interface model, is presented in Section III-D.

In the third stage, our attention turns to the analysis of overloaded operational signatures and their combinations resulting in *service variants*, i.e. various ways of invoking a service. The goal is to derive from API specifications all valid service variants and capture them via so-called *subtyping* relation between business entities. Subtypes of business entities are prevalent in enterprise systems and they may be arbitrarily nested in a type inheritance (or specialisation) hierarchy, leading to complex structures. Given an API specification, service variants are essentially a set of possible combinations of input parameters of operations, which can be transformed into subtypes of business entities. Hence, by extending a business entity model (obtained from the first stage) with subtyping relations, it can be used to specify service variants. Figure 1 (c) depicts an abstract example of a business entity model with subtyping, where business entity A has three subtypes A_1, A_2, and A_3, and furthermore, entity A_2 has two subtypes A_{21} and A_{22}. The details of service variants analysis are discussed in Section III-E.

C. STRUCTURAL INTERFACE ANALYSIS

We propose a structural interface analysis method, which can be used to systematically extract from an API specification the ('hidden') business entities and their relations to form a business entity model. Below, we formally define the concepts of API specification, business entity, business entity relations, and business entity model. The definitions serve as a necessary preliminary for the design of our method and algorithm to derive business entity models from API specifications. Also, the FedEx Open Shipping service WSDL specification shown in Listing 1 is used to illustrate the formal definitions.

Definition 1 (API Specification): An API specification s is a tuple $(OP, P, \kappa, \gamma, \xi^P, \lambda^P)$. OP is the set of operations and P is the set of parameters. $\forall p \in P, \forall op \in OP$, $\kappa : P \times OP \rightarrow \{input, output, na\}$ indicates if p is an input or output parameter of operation op, or p is not associated (na) with op. $\gamma : P \rightarrow \{primitive, complex\}$ specifies whether each $p \in P$ is a primitive or complex parameter. $P_C = P|_{\gamma(p)=complex}$ is the set of complex parameters in P. $\xi^P \subseteq P_C \times P$ specifies the direct nesting relations between parameters. ξ^P is intransitive (i.e. $\forall_{(p,p')\in\xi^P} \neg\exists_{p''\in P}(p\xi^P p'' \wedge p''\xi^P p'))$ and irreflexive (i.e. a parameter is not nested in itself). $\lambda^P : \xi^P \rightarrow \{mandatory, optional\}$ indicates for each $(p, p') \in \xi^P$ whether p' is a mandatory or optional element nested in p.

Remark: In a WSDL specification, the attribute value of `minOccurs` of a parameter p' within its parent parameter p indicates whether the nesting relation between p and p' is mandatory (`minOccurs>0`) or optional (`minOccurs=0`).

Example: In Listing 1, the *OpenShipService* specification has only one operation *createOpenShipment*, and this operation has only one input parameter, *CreateOpenShipmentRequest*. The type of this input parameter is *CreateOpenShipmentRequest*. It is a complex parameter and has two nested parameters: *Index* and *RequestedShipment*, both being optional (*minOccurs="0"*). *RequestedShipment* is also a complex parameter which further contains 14 parameters, i.e., 14 nesting parameters of *RequestedShipment*.

Definition 2 (Business Entity): Let E be a set of business entities. For each $e \in E$, $\mathcal{N}_e$ is the name of e, key_e the unique identifier of e, and $\mathcal{A}_e$ the finite set of attributes associated with e. Given an API specification $s, f : P_C \rightarrow E$ captures the mapping from a complex parameter $p \in P_C$ to a business entity $e \in E$, and $\forall_{p\in P_C}\forall_{p'\in P_C}((p, p') \in \xi^P \Rightarrow f(p) \neq f(p'))$ (i.e. two nested parameters cannot be mapped to the same business entity). $\xi^E \subseteq E \times E$ specifies the direct nesting relations between business entities, and $\forall_{(e,e')\in\xi^E}\exists_{(p,p')\in\xi^P}(e = f(p) \wedge e' = f(p'))$ (i.e. the nesting relations between business entities are informed by the nesting relations between the corresponding parameters in s). $\lambda^E : \xi^E \rightarrow \{mandatory, optional\}$ indicates, for each $(e, e') \in \xi^E$, whether e' is a mandatory or optional element of e, and $\lambda^E(e, e') = \lambda^P(p, p')$ if $e = f(p)$ and $e' = f(p')$.

Example: Assume that the *RequestedShipment* parameter is mapped to business entity *ShippingOrder*. As aforementioned, *RequestedShipment* contains 14 parameters, and accordingly *ShippingOrder* has 14 attributes. Assume that the *RequestedShipment* further contains *RequestedShipper* which is mapped to business entity *Shipper*. Then, *Shipper* is nested in *ShippingOrder* because *RequestedShipper* is a nesting parameter of *RequestedShipment*.

Definition 3: (Domination, *adapted from* [10]) Given an API specification s and a set of business entities E, for two business entities e, e' in E where $\exists\, p \in P_C\; \exists\, p' \in P_C$ s.t. $e = f(p)$ and $e' = f(p')$ (i.e. both e and e' are derived from s), e dominates e', denoted as $e \mapsto e'$, if: (1) $\forall\, op \in OP$, $\kappa(p', op) = input \Rightarrow \kappa(p, op) = input$; and (2) $\exists\, op \in OP$ s.t. $\kappa(p, op) = input$ and $\kappa(p', op) \neq input$.

Remark: Domination is defined between business entities and is derived from how the corresponding parameters associate with each other in an API specification. Assume business entity e is mapped from parameter p and e' from p'. If every operation in the service interface specification that has p' as an input parameter must also have p as an input parameter, whereas at least one operation that has p as an input parameter does not need to have p' as an input parameter, then the corresponding business entity e dominates e'. Domination is defined to assist in the definitions of the Exclusive and Inclusive Containment relations below.

Example: As aforementioned, the *RequestedShipment* parameter is mapped to business entity *ShippingOrder*. Also, consider that the *RequestedPackageLineItem* parameter is mapped to business entity *ShipmentLineItem*. Assume that every operation (e.g., *modifyPackageInOpenShipmentRequest*) that requires *ShipmentLineItem* also needs *ShippingOrder*, while there is at least one operation (e.g *createOpenShipment*) that requires *ShippingOrder* but does not need *ShipmentLineItem* (e.g., because a *ShippingOrder* can be created without a *ShipmentLineItem*). Then, *ShippingOrder* dominates *ShipmentLineItem*.

Definition 4 (Exclusive and Inclusive Containment): Given an API specification s, a set of business entities E and their nesting relations ξ^E, $E_s = \{e \in E | \exists_{p\in P_C}\, (e = f(p))\}$ is the set of business entities derived from s, and $\xi^E_s = \{(e, e') \in \xi^E | \exists_{(p,p')\in\xi^P}(e = f(p) \wedge e' = f(p'))\}$ specifies the entity nesting relations derived from s. Then, $\omega_s = \{(e, e') \in \xi^E_s | e \mapsto e' \wedge \neg\exists_{e''\in E_s} e'' \mapsto e'\}$ defines the exclusive containment relations between business entities in E_s, and $\varphi_s = \xi^E_s \setminus \omega_s$ specifies the inclusive containment relations between them.

Example: Assume that the *ShipmentLineItem* entity is nested in *ShippingOrder* entity and the latter is the only business entity that dominates the former. Then, the *ShippingOrder* entity exclusively contains *ShipmentLineItem*. Next, as aforementioned, the *Shipper* entity is nested in *ShippingOrder*, and also assume that the latter does not dominate the former. Then, *ShippingOrder* entity inclusively contains *Shipper*. Furthermore, if it is mandatory that *Shipper* is nested

in *ShippingOrder*, then the relationship between the two entities is mandatory Inclusive containment.

Definition 5 (Business Entity Model): A business entity model m derived from an API specification s is a tuple (E_s, ξ_s^E, ω_s, φ_s, λ^E). It consists of the set of derived business entities E_s, their nesting relations ξ_s^E which are further divided into exclusive containment relations ω_s and inclusive containment relations φ_s, and λ^E specifying the mandatory or optional attribute of a nesting/containment relation.

Next, we propose a three-step method (see Algorithm 1) to derive business entity models for (complex) APIs specified in interface description languages such as WSDL.

The first step (*lines 4-16*) is to map parameters to business entities and their attributes using semantic matching techniques. A key task is the derivation of business entities from complex parameters. It is carried out by searching in a repository of business entities ($\mathcal{R}$) for one (e) that semantically matches a given complex parameter (p), where the repository $\mathcal{R}$ is a collection of pre-identified business entities based on domain-specific knowledge. Users can designate an ontology for a particular context at design time. This ontology is stored in R, and the complex parameter is checked against the repository to determine if there is a matching entry in it. A number of existing semantic matching approaches with tool support (e.g., COMA++,[4] SimMetrics[5]) can be applied. To measure the semantic similarity between a parameter and an entry in the predefined ontology, this research adopts COMA++, a tool that applies several different semantic matching algorithms and provides an interactive and iterative match process in which users can decide whether to confirm or reject a proposed match based on matching results. The matching operation takes two schemas as inputs, and produces a mapping between elements of these two resources. The tool uses a variety of measures to calculate the similarity between two schema elements or ontology concepts. The similarity confidence is measured by a float number between 0 and 1, where the former denote entirely different (strong dissimilarity) and the later denotes largely similar (strong similarity).

In our algorithm, this search process is captured by function SemanticMatch(p, $\mathcal{R}$) (line 5) which either returns a business entity that semantically matches p or an empty element (*null*) when no match can be found. Next, if a matching entity e is retrieved, the mapping between e and the corresponding parameter p is recorded (line 11), and all the parameters p' nested in p are mapped to the attributes of e (lines 12-14). Mapping a parameter to an attribute is captured by function ConvertToAttr(p') (line 13).

The second step (*lines 18-24*) is to derive nesting relations between business entities. By examining each pair of nested parameters, the algorithm checks whether there are semantically matching business entities for both parameters, and if so, it records the pair of two business entities as nested entities (line 20). At the same time, the value of a parameter nesting relation being mandatory or optional is also mapped to that of the corresponding entity nesting relation (line 21).

[4] http://dbs.uni-leipzig.de/de/Research/coma.html
[5] https://github.com/Simmetrics/simmetrics

Algorithm 1 DeriveBEModel

input: API specification s /* ($OP, P, \kappa, \gamma, \xi^P, \lambda^P$) */
1: $E_s, \xi_s^E, F, \lambda^E := \varnothing$
2: **for** each $op \in OP$ **do**
3:   */* Step 1: Map parameters to business entities and attributes */*
4:   **for** each $p \in P$ s.t. $\kappa(p, op) \neq na \wedge \gamma(p) = complex$ **do**
5:     $e :=$ SemanticMatch(p, $\mathcal{R}$)
6:     **if** $e \neq null$ **then**
7:       **if** $e \notin E_s$ **then**
8:         $E_s := E_s \cup \{e\}$
9:         $\mathcal{A}(e) := \varnothing$
10:       **end if**
11:       $F := F \cup \{(p, e)\}$
12:       **for** each $p' \in P$ s.t. $p \, \xi^P p'$ **do**
13:         $\mathcal{A}(e) := \mathcal{A}(e) \cup \{$ConvertToAttr$(p')\}$
14:       **end for**
15:     **end if**
16:   **end for**
17:   */* Step 2: Derive nesting relations between business entities */*
18:   **for** each $(p, p') \in \xi^P$ **do**
19:     **if** $\exists(p, e) \in F \wedge \exists(p', e') \in F$ **then**
20:       $\xi_s^E := \xi_s^E \cup \{(e, e')\}$
21:       $\lambda^E := \lambda^E \cup \{(e, e'), \lambda^P(p, p')\}$
22:     **end if**
23:   **end for**
24: **end for**
25: */* Step 3: Refine into exclusive and inclusive containment relations */*
26: $\omega_s := \xi_s^E$
27: $\varphi_s := \varnothing$
28: **for** each $(e, e') \in \xi_s^E$ **do**
29:   **if** CheckDomination(e, e') **then**
30:     $E := E_s \setminus \{e, e'\}$
31:     **while** $E \neq \varnothing$ **do**
32:       select $e'' \in E$
33:       **if** CheckDomination(e'', e') **then**
34:         $E := \varnothing$
35:         $\omega_s := \omega_s \setminus \{(e, e')\}$
36:         $\varphi_s := \varphi_s \cup \{(e, e')\}$
37:       **else**
38:         $E := E \setminus \{e''\}$
39:       **end if**
40:     **end while**
41:   **end if**
42: **end for**
43: **return** $(E_s, \xi_s^E, \omega_s, \varphi_s, \lambda^E)$

The third step (*lines 26-41*) is to refine business entity nesting relations into exclusive and inclusive containment relations. Initially, the algorithm assumes that all the nesting relations are exclusive containment relations (*lines 26-27*). Then, for each pair of nested entities, it checks whether one (parent entity e) dominates the other (child entity e') as captured by function CheckDomination(e, e') (*line 29*). If so, the algorithm continues to inspect one by one the remaining entities (e'') and as soon as it discovers an entity e'' that dominates the above child entity e', the pair of nesting relation (e, e') is removed from exclusive containment relations to inclusive containment relations (*lines 30-40*). After this third step, the business entity model of the given API is derived as the output of Algorithm 1.

Finally, the run-time complexity of Algorithm 1 is calculated by taking into account the complexity of all the contained loops and their nested loops as follows. There are two top level loops: *loop-1* from line 4 to line 24, and *loop-2* from line 28 to line 42. For *loop-1*, the size of the input is $|OP|$, and it has nested loops to implement the first and second steps. In the first step (*lines 4-16*), the (maximum) size of the input for each loop is: $|P|$ at the first loop (lines 4-16), and $|P| - 1$ at its nested loop (lines 12-14). In the second step (*lines 18-24*), the size of the input for the loop (lines 18-23) is $|P|$. For *loop-2*, the size of the input is $|\xi^P|$, and it has one nested loop (lines 31-40) of which the (maximum) size of the input is $|P| - 2$ (considering the most complicated scenario where E_s is the set of business entities derived from all parameters in P). Therefore, the complexity of Algorithm 1 is $\mathcal{O}(|OP| \cdot (|P|)^2 + |\xi^P| \cdot |P|)$.

D. BEHAVIOURAL INTERFACE SYNTHESIS

The business entity model derived from an API specification via structural interface analysis can be used to synthesise service behavioural interfaces, that is, the temporal ordering of operations across multiple business entities. We propose a three-phase method for behavioural interface synthesis and an overview of the method is shown in Figure 2.

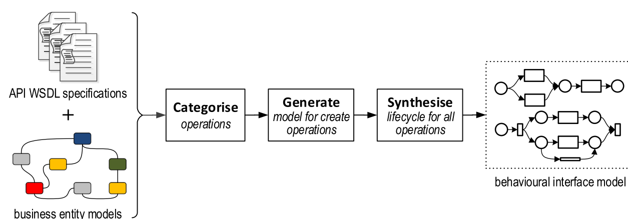

FIGURE 2. Overview of behavioural interface synthesis method for APIs.

1) CATEGORISING CRUD OPERATIONS

At first, the operations associated with the business entities are analysed and categorised into CRUD (i.e. create, read, update, and delete) operations. To launch an instance of business entity e, a *create* operation is invoked requiring attributes of e as input parameters, and upon the invocation it returns a reference to e (referred to as $key(e)$). To retrieve an instance of e, a *read* operation is involved requiring $key(e)$ as an input parameter, and upon the invocation it returns values of attributes of e. Similarly, an *update* operation is for updating an instance of e, of which the invocation requires $key(e)$ and the new values of the relevant attributes of e; and a *delete* operation is for deleting an instance of e.

2) GENERATING MODEL FOR CREATE OPERATION

The second phase focuses on generating behavioural models for *create* operations. These models represent the temporal order of the operations invoked for the creation of a new instance of a business entity, as derived from business entity relations. Here are some examples of the derivation rules. An exclusive containment relation between business entities A and B indicates that A has an exclusive ownership of B. As a result, an instance of B should be launched either as part of or after creating an instance of A. If B is a mandatory part of A (i.e. mandatory exclusive containment), an instance of B must be launched upon or after creation of an instance of A. Next, an inclusive containment relation between B and C indicates that B has an inclusive ownership of C while C has its own independent existence, meaning that launching an instance of C does not necessarily rely on the existence of B.

Let us revisit the excerpt of FedEx Open Shipping service's WSDL specification in Listing 1 (Section II). A Shipping Order exclusive contains Package Line Item(s) and it is a mandatory containment. Hence, a Package Line Item should be only created either as part of creating a Shipping Order or after a Shipping Order is created. Next, a Shipping Order inclusively contains a Shipper and it is mandatory, so a Shipper may exist before the creation of a Shipping Order. A Shipping Order also inclusively contains a Shipping Label, which is optional, and thereby a Shipping Order and a Shipping Label may be created independently.

Algorithm 2 specifies the derivation rules for generating a *behavioural interface model* for *create* operations in general. At first, we define the notion of a behavioural interface model. As specified in Definition 6, such a model is defined as a Petri net, which is a mathematical modelling language for precisely describing the behaviour of distributed systems involving choice, iteration, and particularly concurrent executions. A Petri net consists of places (Q), transitions (T), and flows (F). Transitions are used to model tasks or actions of which the executions often change the state of a system. Places represent pre-conditions required for a task or action to occur as well as post-conditions upon the occurrence of the task or action. Flows capture directed execution order from places and transitions and vice versa. A Petri net is mathematically defined and also offers a graphical notation (e.g., places are drawn as circles, transition as rectangle, flows as directed arcs). Readers interested in Petri nets can find more details in [25].

Definition 6 (Behavioural Interface Model): A behavioural interface model p is a Petri net (Q, T, F) where:

- T is a set of transitions capturing CRUD operations (referred to as *operation transitions*) and non-CRUD operations (referred to as *silent transitions*)[6],
- Q is a set of places specifying the pre- and post-conditions of each of the operations, and
- $F \subseteq (Q \times T \cup T \times Q)$ a set of flow relations that connect a pre-condition to an operation or an operation to a post-condition.

An elementary behavioural interface model p^* consists of at least one input place (q_i), one start transition (τ_i), one pre-condition place (q_{pre}), one operation transition (t), one post-condition place (q_{post}), one end transition (τ_o), and one output place (q_o). An operation transition capturing a *create* operation for business entity e is denoted as t_e^c.

Algorithm 2 constructs a behavioural interface model p_e for creation of business entity e by iteratively performing the derivation for all the business entities that are inclusively and exclusively contained in e. After the initialisation (lines 1-8), the algorithm performs computation in four steps. The first step (lines 9-10) derives from mandatory inclusive containment relations, the second step (lines 11-18) maps the create operations to the corresponding operation transitions, the third step (lines 19-20) deals with both mandatory and optional exclusive containment relations, and the last step (lines 21-22) handles optional inclusive containment relations. Each of the functions to derive from a containment relation is a recursive function that incrementally computes the behavioural interface model p_e (for business entity e) when traversing the containment relations in the given business entity model m (involving e). The resulting behavioural interface model p_e captures the sequences of invoking the operations related to the creation of entity e.

Algorithm 2 consists of a number of linear operations, applies three other algorithms (lines 10, 20 and 22) for generating parts of the overall behavioural interface model that can be derived from the corresponding containment relations between business entity models, and invokes them in a sequential order. Its run-time complexity is calculated by taking into account the complexity of each of those three algorithms. Two algorithms on lines 10 and 22 deal with inclusive containment relation and have the complexity of $\mathcal{O}(|\varphi|)$, the algorithm on line 20 handle exclusive containment relation and has the complexity of $\mathcal{O}(|\omega|)$. Therefore, the complexity of Algorithm 2 is $\mathcal{O}(2|\varphi| + |\omega|)$.

For an abstract demonstration of Algorithm 2, Figure 3 (a) depicts a business entity model comprising e_1 the main business entity, e_2 exclusively contained in e_1 (mandatory), e_4 inclusively contained in e_2 (mandatory), and e_5 inclusively contained in e_2 (optional). Figure 3 (b) shows the corresponding behavioural interface model generated by Algorithm 2 for creating the business entities in Figure 3 (a).

[6]Note that silent transitions are used to capture those operations or actions that are not the focus of our study, and they are needed in a behavioural interface model for specifying the overall execution behaviour.

Algorithm 2 GenerateBIModelForCreateOP

Input: business entity model m /* $(E, \xi^E, \omega, \varphi, \lambda^E)$ */
business entity e /* $e \in E$ */

1: /* *initialise the behavioural interface model* p_e */
2: $p_e := (Q_e, T_e, F_e)$
3: /* *an initial set of places in* p_e */
4: $Q_e := \{q_i^e, q_o^e\}$
5: /* *an initial set of transitions in* p_e */
6: $T_e := \{\tau_i^e, \tau_o^e\}$
7: /* *an initial set of flow relations in* p_e */
8: $F_e := \{(q_i^e, \tau_i^e), (\tau_o^e, q_o^e)\}$
9: /* *Step 1: Derive from mandatory inclusive containment* */
10: $p_e :=$ deriveMandaIncluContainment(m, e, p_e)
11: /* *Step 2: Process the creation* */
12: $t_e^c :=$ ConvertToTransition(op_e^c)
13: **if** $t_e^c = \perp$ **then**
14: **return** *nil*
15: **end if**
16: $Q_e := Q_e \cup \{q_{pre}^e\} \cup \{q_{post}^e\}$
17: $T_e := T_e \cup \{t_e^c\}$
18: $F_e := F_e \cup \{(q_{pre}^e, t_e^c), (t_e^c, q_{post}^e)\}$
19: /* *Step 3: Derive from exclusive containment* */
20: $p_e :=$ deriveExclusiveContainment(m, e, p_e)
21: /* *Step 4: Derive from optional inclusive containment* */
22: $p_e :=$ deriveOptIncluContainment(m, e, p_e)
23: **return** p_e

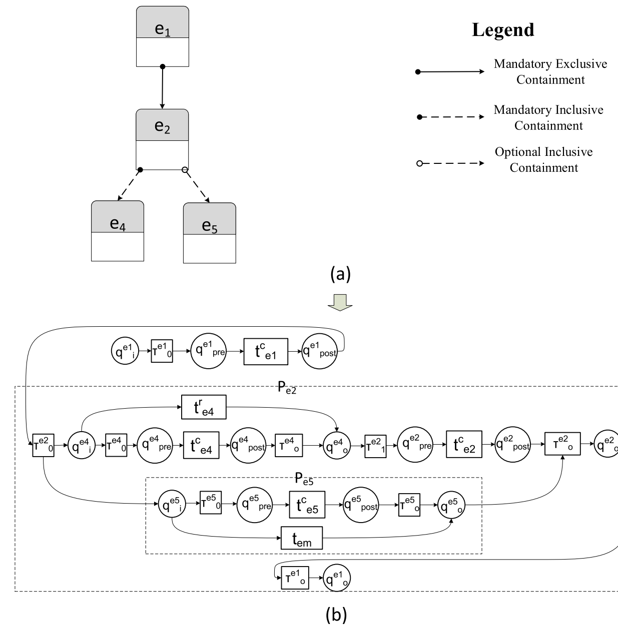

FIGURE 3. An abstract demonstration of Algorithm 2.

3) SYNTHESISING LIFECYCLE FOR ALL OPERATIONS

Finally, to capture the behaviour of invoking the relevant operations, an overall behavioural interface model can be

synthesised representing the life cycle of business entities and the associated operations. With CRUD operations, the notion of *state* can be introduced, and a business entity generally has four states: *ready*, *created*, *updated* and *deleted*, in its life cycle. A lifecycle model specifies the possible ways that a business entity can evolve from an initial state to a final state. Among these states, *ready* is defined as the very initial state of a business entity, indicating it is ready for the entity to be created. A business entity can be created if and only if it is the ready state, and it may be updated or deleted only after it is created. Accordingly, the behavioural interface derivation method yieds two forms of models: business entity creation model and lifecycle model. These models, presented as Petri nets, capture the sequences of operations related to the manipulation of business entities (such as the steps of creating a shipment order or the life cycle of operating on a shipment order), and thus they can be used to inform service users about the invocation sequences that they should adhere when invoking a service via an API.

E. SERVICE VARIANTS ANALYSIS

The business entity model derived from an API specification via structural interface analysis can also be used to derive service variants, that is, various ways of invoking a service. By introducing a subtyping relation between business entities, the problem of deriving service variants can be related to finding subsets of parameters corresponding to business entity subtypes in API operations. We proposed an efficient technique for traversing parameter sets and finding valid subtype invocations, using a Monte Carlo method [26], based on likelihood-free Bayesian sampling. The technique exploits close proximity of parameters in each operation to determine the most likely next parameter to find for a subset based on a previous parameter probabilistic tree search. We herein give a short introduction to this service variant analysis technique proposed in our previous publication [14], where readers interested in the technique can find more relevant details.

Figure 4 depicts an overview of our service variant analysis technique (using an abstract example). A service variant is a combination of input parameters that are accepted when invoking an operation. Given a list of input parameters of an operation and a known service variant (e.g. $op_1(p_1, p_2, p_4)$), the method first initialises a tree with minimum number of leaves. A node of the tree not only stores an input parameter but also the probability of the parameter being a successor of another. With the initial tree, the method searches for other service variants. The key action of the search is to identify the likeliest successor through a Monte Carlo search that employs Bayesian updates and Importance Sampling [26].

The search process takes as input the current parameter being processed, the current path (consisting of a number of parameters traversed along the tree), the current tree node, and a transition kernel variance. It recursively draws a single random variable (i.e. a potential succeeding parameter) based on probability distributions over the current node's child parameters. The search terminates when it reaches the last

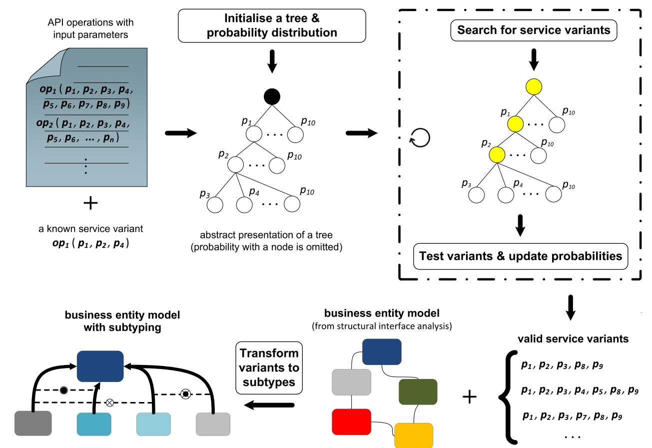

FIGURE 4. Overview of service variant analysis technique.

parameter, and the path drawn from the search is tested thereafter. The test of a path is done via invoking the corresponding operation with the parameters on the path, e.g. invoking op_1 with the sequence of parameters p_1, p_2, p_3, p_8, p_9 shown in Figure 4. If the combination of parameters is accepted, the search then recursively updates the probabilities associated with each of the parameters along the path. If it is not accepted, the entire attempt is ignored and the algorithm proceeds to the next search.

Once a service variant is derived from the above search process, it is then transformed to a subtype of a business entity in the business entity model obtained from structural interface analysis. Recall that a service variant is specified as a set of parameters. The transformation is mainly carried out in three steps: firstly, to search for a business entity e in the business entity model such that each parameter of a service variant v can be mapped to an attribute of business entity e; secondly, to create a subtype entity e_s of e; and thirdly, to map all the parameters of v to the attributes of e_s.

Let us consider the example depicted in Figure 1 (a) and (b). In Figure 1 (a), four business entities are derived from operation op_1 of API specification s. Business entity A has five attributes a_0 to a_4, among which a_3 and a_4 are mapped from complex parameters p_5 and p_{11}, respectively. Parameters p_5 and p_{11} are further mapped to business entities B and D, and in principle the attributes of these two entities are also attributes of object A. Similarly, the attributes of entity C are also attributes of entities B and A. As a result, business entity A has in total 12 attributes a_0 to a_{11} (corresponding to parameters p_2 to p_{13} of operation op_1). Next, we assume that the following five service variants have been obtained: $v_1 = \{p_2, p_3, p_7\}$, $v_2 = \{p_3, p_4, p_7, p_9, p_{10}\}$, $v_3 = \{p_4, p_6, p_{12}, p_{13}\}$, $v_4 = \{p_3, p_4, p_9\}$, and $v_5 = \{p_7, p_{10}\}$. These service variants are used to form the subtype entities shown in Figure 1 (b). Let us start with variant v_1. Three parameters p_2, p_3 and p_7 are mapped to three attributes a_0, a_1 and a_6, respectively. These attributes are a subset of the attributes of entity A, and thus form subtype entity A_1 (of A).

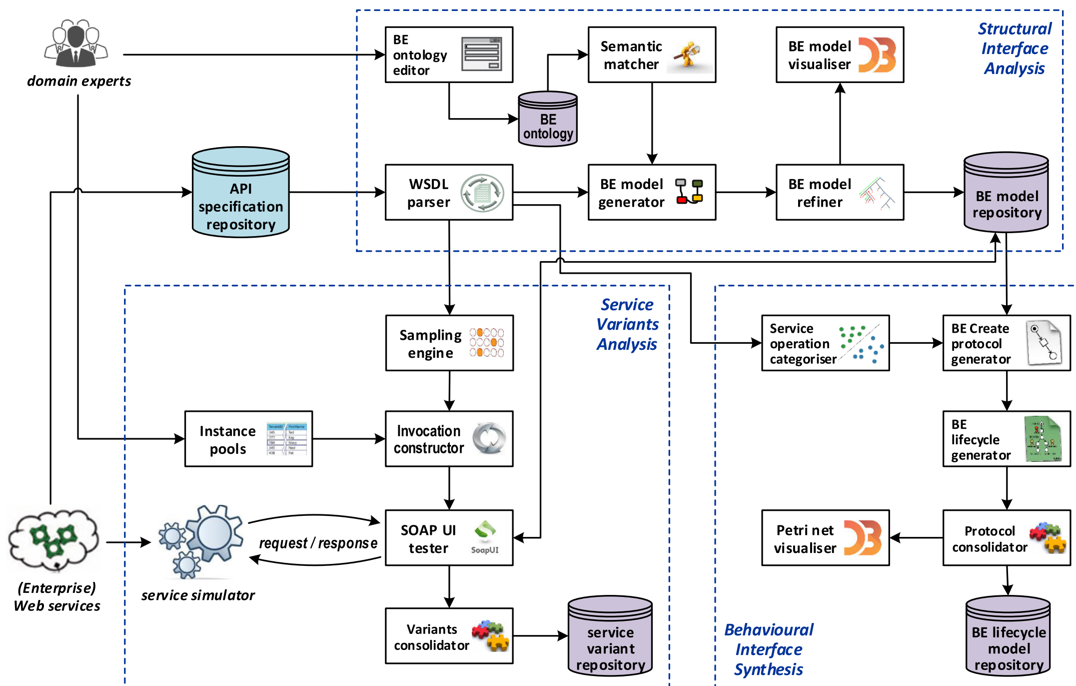

FIGURE 5. Structure of the service interface analyser.

Similarly, v_2 is used to form A_2, v_3 to form A_3, and v_4 and v_5 to form A_{21} and A_{22} (subtypes of A_2), respectively.

We also specify relations between entity subtypes as inspired by subtype exclusion and exhaustion constraints in Object Role Modelling [27]. As shown in Figure 1 (b), a collective exhaustive relation between three subtype entities A_1, A_2 and A_3 indicates that all the attributes of business entity A can be obtained as the union of the attributes of its three subtypes. An exclusive relation between subtypes A_1 and A_3 means that the two entities do not share any common attribute. Accordingly, an exclusive and collective exhaustive relation holds between subtypes A_{21} and A_{22}.

IV. EVALUATION

This section focuses on demonstration and validation of the approach presented in the previous section. A prototypical implementation of our approach, known as the Service Interface Analyser, has been developed (using Java) and released as an open-source tool.[7] The experiments discussed in this section were performed on QUT HPC.[8]

A. TOOL STRUCTURE

The Service Interface Analyser is divided into three modules as shown in Figure 5. Below, we discuss these three modules and the details of each of the components in the tool's structure can be found on the Service Interface Analyser's page on Github.

The *structural interface analysis module* takes API WSDL specifications as input, together with the knowledge of business entity semantics (e.g. based on the input from domain experts), and yield business entity models (BE models for short). Existing API specifications (which are often complex and overloaded) can be retrieved from the *API specification repository*. The output business entity models capture simplified representations of complex structural interfaces by deriving business entities and their relations, and are stored in the *BE model repository*.

Next, the *behavioural interface synthesis module* takes the above BE model as a key input and generates valid sequences of operations. The results are presented as behavioural interface models involving entity creation and ultimately business entity life cycle (BE lifecycle for short). The BE model and BE lifecycle model constitute a simplified presentation layer rendering business entities aligned APIs.

In addition, API WSDL specifications and BE models are also input to the *service variant analysis module* for deriving business entity subtyping relations and also service variants which are stored in the *service variant repository*.

B. VALIDATION OF STRUCTURAL INTERFACE ANALYSIS

The source data for the experiments of structural interface analysis were taken from three categories: Internet Services (IS), Software-as-a-Service (SaaS), and Enterprise Services (ES); while the complexity of their APIs increases from IS,

[7]The source code of the tool's back end is available on `https://github.com/fuguowei/ServiceIntegrationAccelerator`, and that of the front end is on `https://github.com/fuguowei/SIAFrontEnd`.

[8]The Queensland University of Technology high-performance computing lab, see `http://www.itservices.qut.edu.au/researchteaching/hpc/`.

TABLE 1. Structural interface analysis experiment results of 13 selected services specified in the following measures: operations each service provides (N), input parameters (per operation) (N_{IP}), output parameters (N_{OP}), business entities derived (N_{BE}), exclusive containment pairs (N_{ECP}), inclusive containment pairs (N_{ICP}), and false positive rate (F_{PR}).

Service Name (Category)	N	N_{IP}	N_{OP}	N_{BE}	N_{ECP}	N_{ICP}	F_{PR} (%)
Weather Forecast (IS)	2	2	5	0	0	0	0
Find People (IS)	3	2	1	0	0	0	0
MailBox Validator (IS)	1	2	6	0	0	0	0
Amazon S3 (SaaS)	16	9	4	0.56	0.25	0.25	11
Amazon EC2 (SaaS)	157	4	8	2	1	1	20
Amazon Advertising (SaaS)	9	24	243	4	3	1	2
Amazon Mechanical (SaaS)	44	11	271	3	2	1	10
FedEx Ship (ES)	5	709	239	34	40	8	28
FedEx Pick up (ES)	3	137	41	25	23	8	5
FedEx Return (ES)	1	20	15	3	1	0	0
FedEx Close (ES)	6	47	18	4	3	1	12
Open Shipping (ES)	22	309	575	11	9	3	24
Address Validation (ES)	1	31	51	5	3	0	0

SaaS to ES. Altogether 13 widely-deployed services were drawn from *xmethods* (Weather Forecast,[9] Find People,[10] and MailBoxValidator[11]), *Amazon* (S3,[12] EC2,[13] Advertising,[14] and Mechanical[15]), and *FedEx*.[16] Totally 272 operations, 12962 input parameters, and 29700 output parameters were analysed by the Service Interface Analyser. Domain experts were then asked to examine the analysis results, identify false positives, and make any necessary adjustments.

According to the results in Table 1, Internet Services usually have only a few operations with a handful of parameters. For example, the Weather Forecast service has two operations: "GetCitiesByCountry(Country)" and "GetForecastByCity(City, Country)". Although the Service Interface Analyser can pick up and present the Internet services' parameters for providing guidance on the structural interface of these services, service users do not benefit significantly from the analysis results because of their simple APIs.

As Table 1 shows, the APIs of the services in the SaaS category present intermediate complexity. The number of operations provided by the four Amazon Web services ranges from 9 to 157, and the average number of input parameters is between 4 and 24. There are around 3 business entities derived per operation. It may appear that service users can cope with this type of service, as the number of input parameters for some operations is not very large, but the number of operations is quite significant and service users may find it difficult to understand the temporal order among these operations. Hence, having a proper structural analysis is essential to derive such order.

Finally, the category of Enterprise Services contains the most complex APIs, which usually have operations with a large of number of input and output parameters. Hence, it is important to reduce the complexity so that service users can understand the APIs. The experiment results of the six FedEx services shown in Table 1 reveal that the corresponding complex API specifications have been provided with simplified representations, which demonstrates the Service Interface Analyser works effectively for enterprise services.

For example, the Open Shipping service has 22 operations and the average number of the input parameters is 309 and that of the output parameters is 575. After the structural interface analysis, on average, 11 entities per operation were derived. One of the FedEx Open Shipping service's operations, "createOpenShipment", has 1336 input parameters and 596 output parameters. By analysing these parameters, 16 key business entities and their relationships were derived (see Figure 6). This dramatically reduces the complexity as users can now readily understand the interfaces by looking at these business entities and their relationships.

C. VALIDATION OF BEHAVIOURAL INTERFACE SYNTHESIS

A total of 9 services drawn from Amazon and FedEx (as those used for structural interface analysis) are used as the source data for the experiments of behavioural interface synthesis. Table 2 lists the results from the experiments.

In the SaaS category, the number of operations provided by the three Amazon web services ranges from 9 to 44. Based on the business entity models generated and the operations provided by these services, 3, 2 and 9 behavioural models were generated for the creation of business entities involved in the Amazon S3, Advertising, and Mechanical services, respectively. The same number of life cycle models for these entities were also derived.

Taking the Amazon S3 service as an example, Figure 7 (a) depicts a Bucket centric business entity model. The produced

[9] www.restfulwebservices.net/wcf/WeatherForecastService.svc?wsdl
[10] www.findpeoplefree.co.uk/findpeoplefree.asmx?wsdl
[11] ws2.fraudlabs.com/mailboxvalidator.asmx?wsdl
[12] s3.amazonaws.com/doc/2006-03-01/AmazonS3.wsdl
[13] s3.amazonaws.com/ec2-downloads/ec2.wsdl
[14] webservices.amazon.com/AWSECommerceService/AWSECommerceService.wsdl
[15] mechanicalturk.amazonaws.com/AWSMechanicalTurk/2013-11-15/AWSMechanicalTurkRequester.wsdl
[16] www.fedex.com/us/web-services/

TABLE 2. Behavioural interface synthesis experiment results of 9 selected services specified in the following measures: operations each service provides (N), business entities (N_{BE}), behavioural models for entity creation (N_{BM}), and lifecycle models (N_{LC}), the time taken (in milliseconds) for generating these models for each service (T). The behavioural models for entity creation and lifecycle are detailed with number of places, transitions, and flows (denoted by $P/T/F$).

Service Name (Category)	N	N_{BE}	$N_{BM}/P/T/F$	$N_{LC}/P/T/F$	T
Amazon S3 (SaaS)	16	9	3/12/9/18	3/22/17/36	166
Amazon Advertising (SaaS)	9	36	2/12/9/19	2/20/16/36	105
Amazon Mechanical (SaaS)	44	132	9/36/27/54	9/60/48/104	552
FedEx Ship (ES)	5	170	2/8/6/12	2/12/10/20	165
FedEx Pickup (ES)	3	75	1/4/3/6	1/8/6/13	68
FedEx Return (ES)	1	3	1/4/3/6	1/4/3/6	61
FedEx Close (ES)	6	24	4/20/17/36	4/20/16/34	206
Open Shipping (ES)	22	242	4/20/15/31	4/40/32/74	295
Address Validation (ES)	1	5	1/4/3/6	1/4/3/6	48

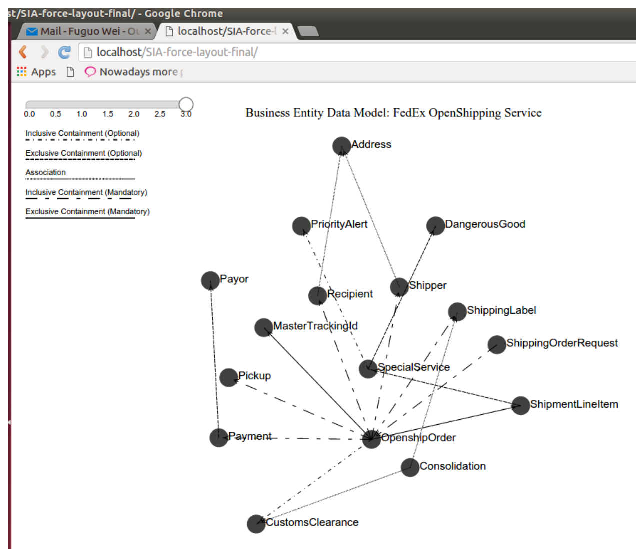

FIGURE 6. A screenshot of the structural interface analysis output of the Fedex Open Shipping service generated by the Service Intergration Accelerator, where each dot represents a business entity and the lines between dots represent the relation between business entities.

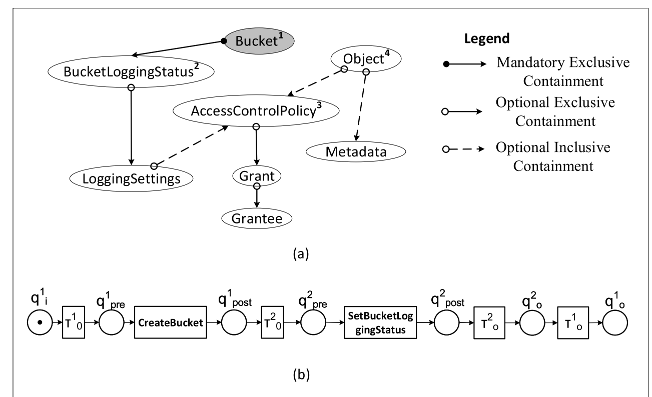

FIGURE 7. The behavioural interface model for the creation of Bucket in Amazon S3.

behavioural interface model for the Bucket's creation is shown in Figure 7 (b). In this model, the transition "Create-Bucket" has been identified as the one that creates an instance of Bucket. Also, entity BucketLoggingStatus is exclusively contained (as mandatory) in entity Bucket, meaning an instance of BucketLoggingStatus has to be instantiated after the creation of a Bucket instance. "SetBucketLoggingStatus" has been identified as the transition that creates an instance of BucketLoggingStatus, so this operation is called after "CreateBucket" as shown in Figure 7 (b).

An enterprise service usually involves numerous business entities and operations. The statistics for the six FedEx services in Table 2 show the number of behavioural interface models generated. For example, by analysing the 22 operations provided by the FedEx Open Shipping service, 4 behavioural models for the creation of 4 business entities (ShippingOrder, ShipmentLineItem, PendingShipment, and Consolidation) were derived. Correspondingly, 4 life cycle models were created for these business entities. The validation of these behavioural interface models was performed by invoking the services with the sequences derived, and the results show that the temporal sequences revealed in the models are valid and match with what is described in the FedEx OpenShipping reference.[17]

D. VALIDATION OF SERVICE VARIANTS ANALYSIS

A challenge for our method is to analyse services and their APIs in the category of Enterprise Services, where the average number of input parameters is around 200. When designing the experiments for service variants analysis, we have simulated services with 20, 50 and 100 parameters, respectively, and with structural complexities comparable to the services analysed. In measuring the performance boost in our service variants analysis method, we have compared it against a brute-force search for problem sizes of 20, 50 and 100 parameters, respectively.

Brute-force search (a.k.a. exhaustive search) is very general problem-solving technique that exhausts all possibilities in order to reach a solution. In the context of deriving service

[17] https://images.fedex.com/templates/components/apps/wpor/secure/downloads/pdf/201507/FedEx_WebServices__DevelopersGuide_v2015.pdf

variants, this method searches all possible service variants in order to identify valid ones. This approach is prohibitive and impractical, especially when the number of parameters is as large as what enterprise services have, because the search space is enormous and simply cannot enumerate all possible parameter combinations.

In the simulated servers, variants were generated at random, so that we could determine the success rates of recovering those variants with our method. In each experiment, the server generated sets of twenty service variants of different lengths and deviations from one another. Experiments in the problem stage of 20 parameters, involved variants selected at random of lengths: 5, 8, 11, 14, and 17; in the problem stage of 50 parameters, the lengths of variants were: 10, 15, 20, 25, 30, 35, and 40; and for the problem stage of 100 parameters, the lengths were: 10, 20, 30, 40, 50, 60, 70, and 80. In creating statistical confidence, two-hundred experiments were conducted for each problem stage, and experiments ran for six days.

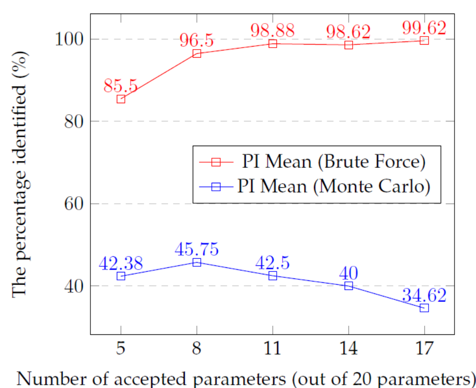

FIGURE 8. The performance comparison between the brute-force and Monte Carlo methods given 20 parameters.

As depicted in Figure 8, the variant analysis method proposed in this study fared worse than the brute-force one when the total number of parameters is 20. On average, the variant analysis method was able to identify from approximately 35% to 46% of a total of 20 valid variants among the 5 sets given (see the blue line in Figure 8). There is a standard deviation for each set. The maximum percentage picked up by the method is 90%, given 5 and 8 known input parameters, and the minimum one is 10% given, 17 known input parameters. Such differences between the results obtained using the brute-force and Monte Carlo methods are due to the fact that the more parameters a variant has, the more difficult it is for the sampling method to identify the variant. The red line in Figure 8 presents the performance of a brute-force method given the first test case, where the method was able to derive the majority of valid service variants, whereas the Monte Carlo sampling could identify only approximately 40 per cent of them. This is because the search space is still within the reach of the capability of the brute-force method and

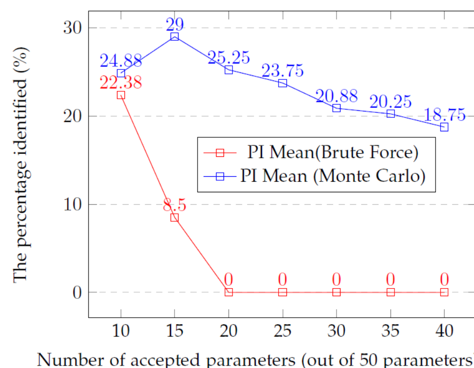

FIGURE 9. The performance comparison between the brute-force and Monte Carlo methods given 50 parameters.

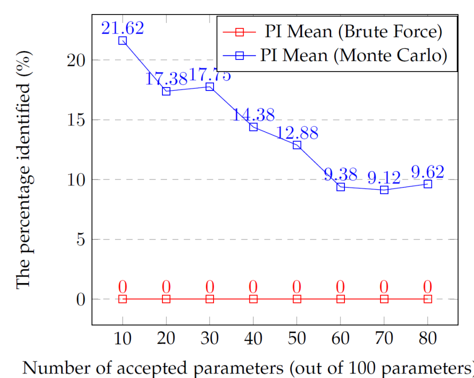

FIGURE 10. The performance comparison between the brute-force and Monte Carlo methods given 100 parameters.

all combinations were traversed by a trial and error method within a sensible amount of time.

However, when the number reached 50, 100, or greater, the brute-force approach became ineffective, while the Monte Carlo search method was more effective by contrast. This can be seen by the performance comparison in Figures 9 and 10. The brute-force method failed to identify anything when the length of the given path was greater than 20 for a total number of 50 parameters (see the red line in Figure 9). In all the experiments, the percentage of the hit rate in the applications of the Monte Carlo sampling method is greater than that of the brute force approach, meaning that the proposed sampling method is more likely to pick up a valid variant. The results showed that the Monte Carlo sampling could find variants in search spaces previously thought to be prohibitive.

In addition, we also evaluated the Monte Carlo-based method on a simulated FedEx shipment service.[18] While this service involved seven operations, the server only simulated the core ''processShipment'' operation. This operation involved 1053 input parameters and 565 output parameters. From which we have derived 34 business entities using the

[18] www.fedex.com/templates/components/apps/wpor/secure/downloads/xml/Aug13/advanced/ShipService_v13.xml

structural interface analysis method (described in Section III-C). From these, a cohesive set within the context of the shipment service of 43 core parameters were selected to demonstrate our method. The search method derived 11 of 20 valid combinations and took 8602 minutes in total given a known combination. An example of service variants of the simulated FedEx Shipment service derived from our approach and presented in the form of business entity subtypes is illustrated in Figure 11.

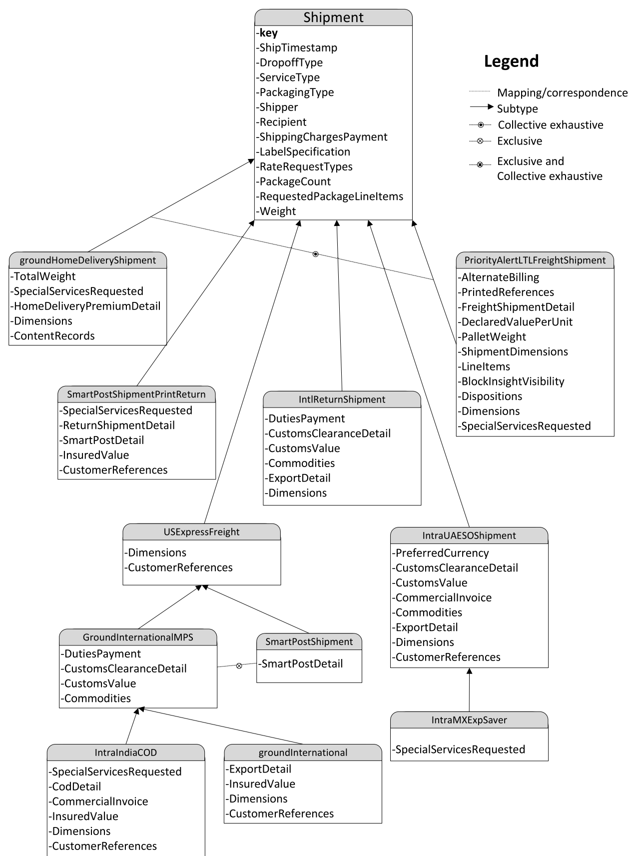

FIGURE 11. Service variants of simulated FedEx Shipment presented in the form of business entity subtypes.

V. RELATED WORK

API analysis is significant, as seen through many commercial products e.g. Microsoft BizTalk Mapper,[19] Stylus Studio XML Mapping Tools,[20] and SAP XI Mapping,[21] and has been the subject of ongoing research over many years. It has been motivated originally by the challenges of systems interoperability through Web services, with the large number of contributions relating to the utilisation of semantic ontologies on Web service specifications to address goals of service discovery [28], adaptation [29] and composition [30]. Both structural and behavioural aspects have been covered in Semantic Web service techniques, with ontologies capturing domain-based entity types and relationships. Moreover, techniques have been proposed to exploit entity subtypes underpinning service variants. For instance, Stollberg and Muth [4] addressed this in the context of service variants in the widely used SAP's Enterprise Resource Planning system while Tosic *et al.* [31] proposed a generalised language, Web Service Offerings Language (WSOL), for service variants. WSOL was specifically applied to annotate different variants and versions of a service, which supports service discovery applications.

Semantically annotated APIs also make it possible to support service behavioural interface derivation [32]. This form of API typically includes preconditions and postconditions, which define a set of requirements and restrictions such as "must have an existing account with this company", and "only US customers can be served" or "a new purchase order will be created". The key limitation of this body of semantic service analysis techniques is the dependency on manual, user effort to design and annotate API specifications, with upgrade costs incurred as ontologies inevitably evolve. To improve the degree of manual effort, information retrieval techniques have been proposed for ontology conception derivation [33] and schema matching between APIs [34].

Another prominent approach to API analysis involves dynamic analysis (data mining) techniques, which focusses on the analysis of API usage data through systems logs. Of these, a number support the derivation of non-functional properties such as average and variances of call frequencies, data transfer sizes, return times, probability of secondary dependencies and other measures [9]. Other techniques focus on functional aspects captured through recorded service interactions including derivation of message types [5] and message correlation [6]. The availability of API usage data also makes it possible for sequences of service interactions, and, thus, the temporal order of service operation invocations - yielding behavioural models of services not typically available API specifications [7], [8]. Nonetheless, dynamic analysis techniques face the practical limitation that not all possible cases and conditions of execution have been covered in a log [35].

In recent years, static API analysis techniques have been developed to analyse operational signatures only (and no other parts of systems implementation such as source code and execution logs). These techniques have been used to assist developers in improving API structures and supporting API translation to new languages. Static analysis of more contemporary REST APIs has been on structural inconsistencies and other issues concerning resources, e.g., anti-patterns construed on the basis of inconsistencies of resource hierarchies [20]. The focus of static analysis techniques on older style procedural service interfaces, seen through WSDL APIs, has been on analysing various properties and problems

[19] https://docs.microsoft.com/en-us/biztalk/core/creating-maps-using-biztalk-mapper
[20] http://www.stylusstudio.com/press/2005-02-08-sleepycat.html
[21] https://wiki.scn.sap.com/wiki/display/XI/Mapping+Concepts+in+SAP+XI

in operational signatures for improving operational cohesion and coupling. Earlier approaches pointed to the need of heuristic search to overcome the combinatorial problems of brute-force analysis of operations [36], [37]. Meanwhile, Bertolino *et al.* [38] analysed API operations to derive their dependencies, which was based on input and output parameter dependencies of the operations using type matching heuristics. This is used to derive the behavioural protocol of the API. The extracted entity models are limited to invocation dependencies and lack structural relationships between entities. Kumaran *et al.* [10] formalised information entities based on the domination theory (utilising co-location heuristics of parameters in operations) and uses this to derive strict containment relationship of entities. Using extracted domination graphs, behavioural models (i.e., state machines) are derived for transitive closures of entities. The only relationship type studied is strict containment by Eshuis and Van Gorp [39]. The work of [15] was one of the first to extract entities from procedural API operation signatures (WSDL based), using a natural language processing technique. Relatively modular operational signatures were used, through which hierarchical relationships between entities were derived, reflecting the hierarchical resource relationships encountered in REST APIs. This reflected the wider goal of the approach, for WSDL to REST API translation. Other text mining approaches for WSDL specifications focus on detection of anti-patterns by Mateos *et al.* [16] and Hirsch *et al.* [17], and quantification of WSDL specification readability by De Renzis *et al.* [18]. Furthermore, a number of metrics were proposed to demonstrate the quality of service interfaces in the context of legacy systems modernization, through the work of Mateos *et al.* [19] which concerned COBOL to SOA migration. However, all these approaches insufficiently treated the problem of overloaded signatures and the challenge of automated service remodularisation.

Service interface remodularization was first addressed by Athanasopoulos and Kontogiannis [15] and Ouni *et al.* [40], based on structural similarity measures of operations in an interface, e.g., similarity of message types used in operations and input/output dependencies of operations. Both approaches analyse structural similarity of operations based and optimization techniques to reason about operations splitting. Athanasopoulos and Kontogiannis [15] focussed on dependencies within operations, operation cohesion, to iteratively split a service interface using a greedy algorithm. Ouni *et al.* [40] focusses on the operation coupling or dependencies across operations. Both approaches are based on traditional algorithms of greedy search and graph partitioning to address this problem. Boukharata *et al.* [41] extended the work of Ouni *et al.* [40] to determine sequential similarity (input/output dependency), communication similarity (message types) and semantic similarity (data types related to domain concepts). These extracted similarity measures are used through multi-objective optimization (using NSGA-II), to find optimal modularization of operations, reaching the best trade-off between minimizing coupling, maximizing cohesion, and minimizing the interfaces modifications.

Our paper extends upon the approach of Kumaran *et al.* [10] and derives a comprehensive entity model by comparison. Specifically, we extend the application of domination theory to operation parameter co-location analysis to derive different entity relationships types, i.e. strict containment, weak containment and basic associations, each with mandatory and optional cardinalities. These allow a more refined understanding of API operation structure allowing recommendations to be made for operation restructure as published in [12], [42]. Our contribution to intra-operational structural analysis, based on probabilistic tree search of parameter space, has led to a novel technique for service variants derivation from API operations and therefore variant-based service restructure recommendations [14]. Using refined relationship insights of entity models, we have also developed entity behavioural models, focussed currently on entity existence operations (i.e. create and delete operations) [13]. For example, if one of an operation's input parameters depends on at least one of another operation's output parameters, determined through the dependency captured through corresponding entities in the entity model, then that operation should be invoked the other one.

We also note there have been contributions to systems analysis using static and dynamic analysis techniques which relate to APIs. For example, automata learning has proven effective in constructing behavioural interfaces of event reactive systems [43]–[45]. This method actively interrogates target systems with queries, observes behavioural models produced in response to the queries, and learns these models using machine learning algorithms. It is important to handle the data dependencies between invocations, so analysis of data parameters and data flows for the derivation of behavioural models can be complemented by the utilisation of automata learning [45]–[48]. For instance, the work of Bertolino *et al.* [38] has been complemented by active automata learning [45] to improve the accuracy of behavioural models derived. By contrast our present paper has focussed on reliance of API code for static analysis to extract, as best as possible, structural and behavioural properties which can support API restructure. This, coincidentally, reflects the reality that APIs are typically decoupled and publicly available that software systems code.

VI. CONCLUSION

Despite the fact that Web-based APIs are complex and overloaded, there is a lack of sufficient knowledge about the structural composition as well as invocation sequences of these interfaces. The research reported in this paper has presented for the first time a systematic approach to yield a simplified and insightful presentation of these complex interfaces without requiring their comprehensive semantics. The approach is composed of three building blocks, which are structural interface analysis, behavioural interface analysis, and service variants analysis.

Future work on structural interface analysis can be seen as follows. In this paper, the concepts of business entities and their containment relationships have been introduced and formalised into a business entity model. Multiplicity, specifying the number of instances of one business entity allowed in a containment relationship with another business entity, leads to iteration (i.e. allowing the creation of multiple instances) in a service behavioural interface. This is to be studied in future. Next, the idea of structural interface analysis involving derivation of business entity models presented in this study can be applied to RESTful service interfaces, which is worth of investigation in future.

With regards to behavioural interface synthesis, a method for deriving state-based behavioural interfaces upon a given business entity model has been proposed in the paper. The introduction of states into service behavioural interfaces enables flexible service interactions. The notion of states enables a declarative mechanism for interaction needs, without prescribing which services or which order of interactions should be taken. For future directions, this opens up the possibility of a dynamically determined execution of interaction, such as the interactions relevant to advancing states, the interactions involved in fulfilling interaction progress, and the interleaving of interactions across different services. Advanced operations, such as cancellations back to previous states and replacements with new providers going forward, also become possible.

Finally, in service variants analysis, a Monte Carlo-based sampling method has been developed to search for service variants. The primary significance of this method is that, through experiments, good results can be produced even in a very large search space. This is in stark contrast to conventional methods such as a brute-force method, which cannot derive any variants given a large search space. Another prominent feature of the method is that, compared with existing studies, it requires minimal human intervention and inputs – only a known path (i.e., an acceptable service variant) – and it can automatically identify other valid service variants. While the Monte Carlo sampling method has sensible performance results, importance sampling is currently the only variance optimisation. Optimising this method by introducing additional mechanisms, such as Markov Blanket [49] remains a future research objective.

REFERENCES

[1] J. Whaley, M. C. Martin, and M. S. Lam, "Automatic extraction of object-oriented component interfaces," *SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 218–228, Jul. 2002.

[2] D. Beyer, A. Chakrabarti, and T. A. Henzinger, "Web service interfaces," in *Proc. 14th Int. Conf. World Wide Web (WWW)*. New York, NY, USA: ACM, 2005, pp. 148–159.

[3] K. Bhattacharya, N. S. Caswell, S. Kumaran, A. Nigam, and F. Y. Wu, "Artifact-centered operational modeling: Lessons from customer engagements," *IBM Syst. J.*, vol. 46, no. 4, pp. 703–721, 2007.

[4] M. Stollberg and M. Muth, "Efficient business service consumption by customization with variability modelling," *J. Syst. Integr.*, vol. 1, no. 3, pp. 17–32, 2010.

[5] T. Nguyen, A. Colman, and J. Han, "A feature-based framework for developing and provisioning customizable Web services," *IEEE Trans. Services Comput.*, vol. 9, no. 4, pp. 496–510, Jul. 2016.

[6] B. Serrour, D. P. Gasparotto, H. Kheddouci, and B. Benatallah, "Message correlation and business protocol discovery in service interaction logs," in *Advanced Information Systems Engineering* (Lecture Notes in Computer Science) vol. 5074, Z. Bellahséne and M. Léonard, Eds. Berlin, Germany: Springer, 2008, pp. 405–419.

[7] H. R. Motahari-Nezhad, R. Saint-Paul, B. Benatallah, and F. Casati, "Protocol discovery from imperfect service interaction logs," in *Proc. IEEE 23rd Int. Conf. Data Eng.*, 2007, pp. 1405–1409.

[8] K. Musaraj, T. Yoshida, F. Daniel, M.-S. Hacid, F. Casati, and B. Benatallah, "Message correlation and Web service protocol mining from inaccurate logs," in *Proc. IEEE Int. Conf. Web Services*, Jul. 2010, pp. 259–266.

[9] S. Dustdar and R. Gombotz, "Discovering Web service workflows using Web services interaction mining," *Int. J. Bus. Process Integr. Manage.*, vol. 1, no. 4, pp. 256–266, 2006.

[10] S. Kumaran, R. Liu, and F. Y. Wu, "On the duality of information-centric and activity-centric models of business processes," in *Proc. 20th Int. Conf. Adv. Inf. Syst. Eng. (CAiSE)*, Montpellier, France, Jun. 2008, pp. 32–47.

[11] C. Bartolini, A. Bertolino, E. Marchetti, and A. Polini, "WS-TAXI: A WSDL-based testing tool for Web services," in *Proc. Int. Conf. Softw. Test. Verification Validation*, Apr. 2009, pp. 326–335.

[12] F. Wei, A. Barros, and C. Ouyang, "Deriving artefact-centric interfaces for overloaded Web services," in *Proc. 27th Int. Conf. Adv. Inf. Syst. Eng. (CAiSE)*, Stockholm, Sweden, Jun. 2015, pp. 501–516.

[13] F. Wei, C. Ouyang, and A. Barros, "Discovering behavioural interfaces for overloaded Web services," in *Proc. IEEE World Congr. Services*, New York, NY, USA, Jun./Jul. 2015, pp. 286–293.

[14] R. Rasmussen, A. Barros, and F. Wei, "A likelihood-free Bayesian derivation method for service variants," *J. Syst. Softw.*, vol. 143, pp. 87–99, Sep. 2018.

[15] M. Athanasopoulos and K. Kontogiannis, "Extracting REST resource models from procedure-oriented service interfaces," *J. Syst. Softw.*, vol. 100, pp. 149–166, Feb. 2015.

[16] C. Mateos, J. M. Rodriguez, and A. Zunino, "A tool to improve code-first Web services discoverability through text mining techniques," *Softw., Pract. Exper.*, vol. 45, no. 7, pp. 925–948, Jul. 2015.

[17] M. Hirsch, A. Rodriguez, J. M. Rodriguez, C. Mateos, and A. Zunino, "Spotting and removing WSDL anti-pattern root causes in code-first Web services using NLP techniques: A thorough validation of impact on service discoverability," *Comput. Standards Interface*, vol. 56, pp. 116–133, Feb. 2018.

[18] A. De Renzis, M. Garriga, A. Flores, A. Cechich, C. Mateos, and A. Zunino, "A domain independent readability metric for Web service descriptions," *Comput. Standards Interface*, vol. 50, pp. 124–141, Feb. 2017.

[19] C. Mateos, A. Zunino, A. Flores, and S. Misra, "COBOL systems migration to SOA: Assessing antipatterns and complexity," *Inf. Technol. Control*, vol. 48, no. 1, pp. 71–89, Mar. 2019.

[20] F. Palma, J. Dubois, N. Moha, and Y. Guéhéneuc, "Detection of REST patterns and antipatterns: A heuristics-based approach," in *Proc. 12th Int. Conf. Service-Oriented Comput. (ICSOC)*, in Lecture Notes in Computer Science, vol. 8831. Paris, France: Springer, Nov. 2014, pp. 230–244.

[21] A. Bennaceur and V. Issarny, "Automated synthesis of mediators to support component interoperability," *IEEE Trans. Softw. Eng.*, vol. 41, no. 3, pp. 221–240, Mar. 2015.

[22] X. Lu, M. Nagelkerke, D. V. D. Wiel, and D. Fahland, "Discovering interacting artifacts from ERP systems," *IEEE Trans. Services Comput.*, vol. 8, no. 6, pp. 861–873, Nov. 2015.

[23] C. Mateos, M. Crasso, J. M. Rodriguez, A. Zunino, and M. Campo, "Measuring the impact of the approach to migration in the quality of Web service interfaces," *Enterprise Inf. Syst.*, vol. 9, no. 1, pp. 58–85, Jan. 2015.

[24] W. Kongdenfha, H. R. Motahari-Nezhad, B. Benatallah, and R. Saint-Paul, "Web service adaptation: Mismatch patterns and semi-automated approach to mismatch identification and adapter development," in *Web Services Foundations*. New York, NY, USA: Springer, 2014, pp. 245–272.

[25] J. L. Peterson, "Petri nets," *ACM Comput. Surv.*, vol. 9, no. 3, pp. 223–252, 1977.

[26] C. P. Robert and G. Casella, *Monte Carlo Statistical Methods* (Springer Texts Statistics). New York, NY, USA: Springer-Verlag, 2005.

[27] T. Halpin, A. Morgan, and T. Morgan, *Information Modeling and Relational Databases* (Morgan Kaufmann Series in Data Management Systems). Amsterdam, The Netherlands: Elsevier, 2008.
[28] M. Malaimalavathani and R. Gowri, "A survey on semantic Web service discovery," in *Proc. Int. Conf. Inf. Commun. Embedded Syst. (ICICES)*, Feb. 2013, pp. 222–225.
[29] K. Mecheri and L. Souici-Meslati, "Semantic interoperability of Web services: A survey," in *Proc. Int. Conf. Mach. Web Intell.*, Oct. 2010, pp. 82–87.
[30] A. L. Lemos, F. Daniel, and B. Benatallah, "Web service composition: A survey of techniques and tools," *ACM Comput. Surv.*, vol. 48, no. 3, 2015, Art. no. 33.
[31] V. Tosic, K. Patel, and B. Pagurek, "WSOL—Web service offerings language," in *Web Services, E-Business, and the Semantic Web* (Lecture Notes in Computer Science), vol. 2512, C. Bussler, R. Hull, S. McIlraith, M. Orlowska, B. Pernici, and J. Yang, Eds. Berlin, Germany: Springer, 2002, pp. 57–67.
[32] F. Howar, B. Jonsson, M. Merten, B. Steffen, and S. Cassel, "On handling data in automata learning," in *Leveraging Applications of Formal Methods, Verification, and Validation* (Lecture Notes in Computer Science), vol. 6416, T. Margaria and B. Steffen, Eds. Berlin, Germany: Springer, 2010, pp. 221–235.
[33] P. D. Bruza, A. Barros, and M. Kaiser, "Augmenting Web service discovery by cognitive semantics and abduction," in *Proc. IEEE/WIC/ACM Int. Joint Conf. Web Intell. Intell. Agent Technol.*, vol. 1, Sep. 2009, pp. 403–410.
[34] C. Drumm, M. Schmitt, H.-H. Do, and E. Rahm, "Quickmig: Automatic schema matching for data migration projects," in *Proc. 16th ACM Conf. Conf. Inf. Knowl. Manage. (CIKM)*, 2007, pp. 107–116.
[35] A. Wasylkowski and A. Zeller, "Mining operational preconditions," Universität des Saarlandes, Saarbrücken, Germany, Tech. Rep., 2008. [Online]. Available: https://www.st.cs.uni-saarland.de/models/papers/wasylkowski-2008-preconditions.pdf
[36] W. Mkaouer, M. Kessentini, A. Shaout, P. Koligheu, S. Bechikh, K. Deb, and A. Ouni, "Many-objective software remodularization using NSGA-III," *ACM Trans. Softw. Eng. Methodol.*, vol. 24, no. 3, pp. 17:1–17:45, 2015.
[37] M. Harman, S. A. Mansouri, and Y. Zhang, "Search-based software engineering: Trends, techniques and applications," *ACM Comput. Surv.*, vol. 45, no. 1, pp. 11:1–11:61, 2012.
[38] A. Bertolino, P. Inverardi, P. Pelliccione, and M. Tivoli, "Automatic synthesis of behavior protocols for composable Web-services," in *Proc. 7th Joint Meeting Eur. Softw. Eng. Conf. ACM SIGSOFT Symp. Found. Softw. Eng. Eur. Softw. Eng. Conf. Found. Softw. Eng. Symp. (ESEC/FSE)*. New York, NY, USA: ACM, 2009, pp. 141–150.
[39] R. Eshuis and P. Van Gorp, "Synthesizing data-centric models from business process models," *Computing*, vol. 98, no. 4, pp. 345–373, Apr. 2016.
[40] A. Ouni, M. Kessentini, K. Inoue, and M. O. Cinneide, "Search-based Web service antipatterns detection," *IEEE Trans. Services Comput.*, vol. 10, no. 4, pp. 603–617, Jul. 2017.
[41] S. Boukharata, A. Ouni, M. Kessentini, S. Bouktif, and H. Wang, "Improving Web service interfaces modularity using multi-objective optimization," *Automated Softw. Eng.*, vol. 26, no. 2, pp. 275–312, Jun. 2019.
[42] F. Wei, A. P. Barros, and C. Ouyang, "Introspective service interface synthesis in business networks," Queensland Univ. Technol., Brisbane, QLD, Australia, Tech. Rep. ePrints 74624, 2014. [Online]. Available: https://eprints.qut.edu.au/74624/
[43] R. Alur, P. Černý, P. Madhusudan, and W. Nam, "Synthesis of interface specifications for java classes," in *Proc. 32nd ACM SIGPLAN-SIGACT Symp. Princ. Program. Lang. (POPL)*. New York, NY, USA: ACM, 2005, pp. 98–109.
[44] H. Raffelt, B. Steffen, T. Berg, and T. Margaria, "LearnLib: A framework for extrapolating behavioral models," *Int. J. Softw. Technol. Transf.*, vol. 11, no. 5, pp. 393–407, Nov. 2009.
[45] M. Merten, F. Howar, B. Steffen, P. Pellicione, and M. Tivoli, *Automated Inference of Models for Black Box Systems Based on Interface Descriptions*. Berlin, Germany: Springer, 2012, pp. 79–96.
[46] M. Shahbaz, K. Li, and R. Groz, "Learning parameterized state machine model for integration testing," in *Proc. 31st Annu. Int. Comput. Softw. Appl. Conf. (COMPSAC)*, vol. 2, Jul. 2007, pp. 755–760.
[47] H. Raffelt, B. Steffen, and T. Margaria, "Dynamic testing via automata learning," in *Hardware and Software, Verification and Testing* (Lecture Notes in Computer Science), vol. 4899, K. Yorav, Ed. Berlin, Germany: Springer, 2008, pp. 136–152.
[48] B. Jonsson, "Learning of automata models extended with data," in *Formal Methods for Eternal Networked Software Systems* (Lecture Notes in Computer Science), vol. 6659, M. Bernardo and V. Issarny, Eds. Berlin, Germany: Springer, 2011, pp. 327–349.
[49] D. Koller and M. Sahami, "Toward optimal feature selection," in *Proc. 13th Int. Conf. Mach. Learn.*, 1995, pp. 284–292.

ALISTAIR BARROS is currently a Professor and the Head of Service Sciences Research, School of Information Systems, Queensland University of Technology (QUT). He has worked extensively with SAP, government departments, and various industry partners. His research interests include the development of novel business design utilising services in different industries, service-based IT architecture, and the technical analysis of systems for identifying and validating new designs.

CHUN OUYANG received the Ph.D. degree. She is currently a Senior Lecturer with the School of Information Systems, QUT. Her Ph.D. research interests include system modeling and verification using formal techniques. In the last decade, she has developed strong research interests in and contributed to the areas of business process modeling and analysis, process execution, and process mining. Her recent research interests include predictive analytics, process automation in the cloud, and microservices.

FUGUO WEI received the Ph.D. degree from QUT. He is currently a Service Integration Specialist and Enthusiastic about all aspects of application integration. His Ph.D. research interests include API analysis and service integration. He is currently with the Super Retail Group, Strathpine, QLD, Australia, and the Queensland University of Technology, Brisbane, QLD, Australia. He has undertaken various roles in both academia and industry, including a software developer (senior Java developer and certified Mulesoft developer), a consulting, and a tech lead. His research interests are Web API analysis, service integration and composition, and microservices architecture.