Abstract—Software service has to adapt to constant changes of requirements and environment, which would give rise to the concentration on service evolution. Among the many reasons for evolution, requirements are regarded as the major driving factor. However, most current work on service evolution focuses on different kinds of tactics to keep the compatibility between services and client applications, few of them treat requirements in priority. In this paper, we thoroughly survey the state-of-the-art studies on service evolution, including basic concepts, types, challenges and approaches; then analyze service evolution from four dissimilar dimensions, and for each one we identify the open problems and inadequacies. Based on our analysis and discovery, we argue that requirements driven service evolution would become an essential and promising research direction.

Keywords - software service evolution; requirements driven

I. INTRODUCTION

Evolution is a process of gradual change that takes place over many generations, or a process of gradual and constant change and development in a particular situation over a period of time [1]. It is a biological terminology initially and has been introduced into software engineering to study certain phenomena in software lifecycle, especially after system is launched.

Software evolution is of great importance in satisfying user requirements under certain changes in environment. Just as Lehman’s laws of evolution [2] state, successful software is condemned to change over time. Starting from the pioneer work of him, software evolution has attracted much interest in the last three decades. Usually, it has been associated with the change of code, module or architecture of a software system. Meanwhile, software development is undergoing a radical shift to service oriented paradigm. In the new paradigm, services are regarded as basic building units and are reused for enterprise application integration and development. However, services are residing in a constant changing Internet environment, addressing diversified user demands, which make it unrealistic for them to stay unchanged, especially when they collaborate with other services to fulfill a more extensive set of functionalities.

Among the many factors that lead to evolution, such as aging, technical progress, change of context, legislation and requirements, requirements change is considered to be the fundamental reason. However, most of the work on service evolution is committed to the compatibility between evolved service and clients at either structural [3] or behavior level [4][5]. And few studies pay enough attention to requirements.

II. EVOLUTION: CONCEPTS AND TECHNIQUES

A. Software Evolution

At the early stage, software evolution is sometimes roughly considered to be a synonym of software maintenance. And later, it has been distinguished as a phase that adapts application software to ever-changing user requirements and operating environment, while maintenance is used to refer to general post-delivery activities in the stage model of [7].

It is widely accepted that continuous change would give rise to evolution in the literatures. Representatively, software evolution is treated as changes in the properties of a software system over time or a process of continuous change applied to a software system that is in a feedback loop with the system itself[8]. It has been associated with the change of code, module or architecture of a software system [9] typically. In addition, studies on reverse engineering at the code and process level have also gain momentum. And currently the latest work is focusing on capturing and handling the change of requirements and operational environment [10][11].

B. Service: Definition and Modeling

In literatures, software service (different from novel services such as platform and infrastructure service) is usually referred to web service, which is designed to address the interoperability across heterogeneous platforms and systems. In definition, it is regarded as a set of functionalities that are exposed through standard interfaces [12][13]. With the advancement of service oriented paradigm, where service is used as basic building unit for application development and integration, the way how it is used (i.e. interaction protocol, which defines the message format, sequence and conditions on message sending/receiving) becomes increasingly important. Given this trend, there come out some definitions that depict service as a set of interactions among software components [14] or between customer(s) and provider(s) [15] in order to fulfill tasks or provide solutions.
In accordance with the definitions that focus on different aspects, there are dissimilar modeling techniques for services. To the best of our knowledge, the well-accepted ways for service description include WSDL, Finite State Machine (FSM) [16] and OWL-S. And they describe service at different levels of abstraction: WSDL takes a structure-based perspective; FSM takes a behavioral viewpoint (interaction), while OWL-S focuses on semantics and QoS (composition). Correspondingly, changes to service vary in different models. For example, evolutionary changes for WSDL service usually include adding/modifying/removing operations and messages while those for FSM service model mainly cover adding/removing transition(s) and state(s).

C. Service Evolution

In general, service evolution can be studied at two different abstraction levels: (a) a few researchers consider the problem in the context of a service network/ecosystem, and apply Genetic Algorithms (GAs) in the ecosystem to simulate the biological evolution process, e.g. [17][18]; (b) meanwhile, the majority concentrate on the evolution of a single service.

[17][18] are examples of the first category. To set up an autonomic service network that could respond to unpredictable situations, D. Miorandi et al. [17] propose a bio-inspired service evolution mechanism which includes five steps to build self-evolving services: (a) representing candidate services; (b) distributed fitness evaluation; (c) selection; (e) generating new candidates; (e) distributed continuous optimization. In a similar spirit, T. Nakano et al. [18] present a framework for developing adaptive and scalable network services, which are implemented by a set of autonomous and self-adaptive agents that interact with each other in distributed environment. An evolutionary adaptation mechanism (replication, migration, and death) is designed to improve agents’ fitness (e.g. response time to a request) and evolve their behaviors. The biggest challenge in this kind of approach is to build and maintain a pool of services and evaluate their fitness without bias in real scenarios.

As for a single service, M. Papazoglou [19] defines service evolution as a continuous process of development of a service through consistent and unambiguous changes. It is expressed through service’s different versions, and the key challenge is the forward compatibility between different versions, which are further explained as [19][20]:

- **Forward compatibility**: a guarantee that an old version of a client application could interpret new operation(s)/message(s) introduced by a service.

Though there is some controversy about the definition of forward and backward compatibility (a new version of a client application can interpret the old operation(s)/messages(s) of a service), e.g. it is just the opposite in [21][22], the key point is the same - keeping existing clients compatible with evolved service.

Based on this starting point, a few work discusses the taxonomy of evolutionary changes [19][23][24], and there are more efforts commit themselves to address the incompatibility using different tactics, including versioning [25][26], design pattern/adaptor [21][24][27] and model/theory/contract [3][28]. In summary, table 1 briefly outlines the subjects, changes and approaches.

<table>
<thead>
<tr>
<th>Subject</th>
<th>Evolutionary changes</th>
<th>Approach</th>
</tr>
</thead>
<tbody>
<tr>
<td>service network</td>
<td>adding/modifying/removing a service</td>
<td>genetic algorithms</td>
</tr>
<tr>
<td></td>
<td>taxonomy of changes</td>
<td>taxonomy</td>
</tr>
<tr>
<td></td>
<td>changes of operations, messages (structural level)</td>
<td>versioning</td>
</tr>
<tr>
<td></td>
<td>change of interaction order, adding/removing interaction(s) (behavior/protocol level)</td>
<td>algorithm/adaptor</td>
</tr>
<tr>
<td></td>
<td>quality-of-service</td>
<td>discussed in [3][23]</td>
</tr>
<tr>
<td></td>
<td>semantics</td>
<td>&quot;just enough&quot; validation / extension points</td>
</tr>
<tr>
<td></td>
<td>impact and effect</td>
<td>impact analysis</td>
</tr>
</tbody>
</table>

1) **Taxonomy of Evolutionary Changes**

M. Papazoglou [19] introduces two types of changes: shallow (confined to services or clients) and deep (leading to cascading and side effects), and proposes a contract-based approach as well as a change-oriented lifecycle methodology to address these issues. As for shallow changes, typically it could lead to mismatches between service and clients at two levels [4]: (a) interface level (i.e. structural) (b) interaction protocol level (i.e. behavior, such as messaging order mismatches).

The majority of work currently focuses on WSDL services. M. Treiber [23] defines the basic elements of web service evolution, a.k.a. evolutionary step: requirements change, interface change, implementation change, and QoS change. Meanwhile, according to S. R. Ponnekanti [24], four types of incompatibility may arise: (message) structure (unexpected) value, encoding (instances of different schema types are not identical even if they have the same structure and values) and semantics (identical syntax but different meanings). Similarly, on the basis of WSDL, [30] discusses the change of WSDL elements in detail: portType, message, types, service, binding and endpoint. It also suggests web service versioning and message convention (transforming messages to a common structure that both sender and receiver can understand) as possible solutions. In addition, QoS and semantics changes have also been discussed initially in [23][24][29].

2) **Versioning**

Versioning is a traditional and practical way to address the incompatibility issue [25][26]. A common trait of them is maintaining multiple versions for a specified interface. A simple naming scheme that appends a date or version stamp to the end of a namespace is suggested to ensure the uniqueness of the version identifier [26]. In the meantime, the provider may declare multiple service versions at the same endpoint or each at an endpoint. At the technical level, it relies heavily on the SOA technology – SOAP, WSDL and UDDI.

The versioning model presented in [31] groups a set of services into a subsystem and assigns them the same version identifier. Even if only one service is changed, all
services within the same subsystem will be tagged with a new version number. As a result, multiple versions of the same subsystem may co-exist. On update, the server side will support the redirection from the old version to the new (compatible) version of the requested service.

Also, versioning is usually used together with design pattern [32][33]. Both [32] and [33] aim at addressing the version management issue in web service system. The former concentrates on the server side and makes some extensions to WSDL and UDDI. WSDL is enhanced to describe the attributes of service versions and UDDI is augmented to use versions in a service directory, a proxy is also designed to dynamically update the client application at runtime. The later work lays emphasis on minimizing the impact of service changes. On the consumer side, the authors design a version-aware client model which is able to provide both consumer-transparent invocations at build-time and dynamic proxy generation at runtime through leveraging JAX-RPC.

3) Design Pattern & Tool (Adaptor)

Design pattern is a widely adopted and effective approach in practice. The idea is to maintain multiple versions of a service on the server side, and provide a proxy that enable dynamic binding and invocation for client applications [29][32][33]. For instance, P. Leitner et al. [29] use service version history and selection strategy to provide transparent end-to-end versioning support based on the Vienna Runtime Environment for Service-Oriented Computing (VRESCO) platform. The platform maintains a registry database which stores the meta-data of all service revisions and provides a rebinding proxy with different rebinding strategies (e.g. periodically or on client requests) to enable clients-transparent invocation on evolved services.

In a similar spirit, P. Kaminski et al. [21] propose a solution in the form of a design technique called Chain of Adapters. In this way, to support service evolution, developers should build an interface adapter between the last and the new version of a specific service each time. Also, D. Frank et al. [27] distinguish between a service interface (public) and its implementation (private). During the evolution process, service interface will stay constant and implementation will be changed. The service interface proxy is able to route client requests for a service (interface) to proper implementation version. Multiple proxies are allowed per service in the case of incompatible versions.

To address possible interface mismatches, M. Dumas et al. [36] suggest an algebra over interfaces and a visual language that allows pairs of provided-required interfaces to be linked through algebraic expressions. They also introduce a tool that executes the algebraic expressions and facilitates message buffering and transformation to enact adaptation logic, to achieve interface adaptation.

B. Benatallah et al. [38] propose developing adapters as an approach based on mismatch patterns (e.g., message ordering, extra message, and missing message mismatch) which capture the possible differences between two interaction protocols. Once the pattern has been identified, the adapter code can be generated automatically. Similarly, H. R. Motahari Nezhad et al. [4] provide semi-automatic support for adapter generation to resolve interface mismatch and deadlock-free interaction incompatibility.

Also, S. H. Ryu et al. [5] study the protocol compatibility using path coverage algorithms based on FSM service model and suggest adapter/ad-hoc protocol as solutions.

In addition, [34] and [35] try to keep client applications being synchronized with evolved services through (semi-) automatic client update. In the framework of [34], the authors at first analyze the delta between different versions of a service and export it into a well- formatted document. Drawing on the clients’ usage history, they then employ a Consumer Code Customizer component to highlight the client code fragments that need to be updated. Also, [35] introduces a framework to resolve interface and behavior mismatches through updating the clients automatically based on compatibility measuring and interface/behavior mapping.

4) Model/Contract/Theory

M. Treiber et al. [37] proposes a Service Evolution Management Framework (SEMF) that relies on an information model (e.g. usage statistics, logging) to manage the web service changes on interface, QoS and interaction pattern. V. Andrikopoulos et al. [28] propose a service specification reference model and introduce the concept of service evolution management, which provides an understanding of change impact, control and tracking. In his Ph.D. thesis [3], V. Andrikopoulos explores service compatibility in depth at structural, behavior and non-functional property level. Based on type and set theory as well as the service specification model, he develops a theory to reason and identify the conditions (i.e. a set of changes) under which services can evolve while preserving compatibility. Also, he introduces a contract-based method (an agreement between providers and consumers that specify the expectations and obligations of both parties) to expand the permitted set of changes and provide more flexibility in evolution.

In [22], an algorithm for assessing the compatibility between two revisions of a service is proposed. In their method, service is represented as a model which contains at least one type(s) and the relation between types. A type is a class (e.g. an UML class) which is comprised of operations and attributes. The changes applied to services include adding/ removing/updating operations, attributes and types. Also, the compatibility assessment includes two phases: model and type compatibility. The authors claim their work to be helpful for service developers to assess the impact of changes and put proper versioning requirements in client implementations to avoid runtime incompatibilities.

5) Other Approaches for Service Evolution

Also, there are some other interesting researches on service evolution. For example, [20][39] discuss two well-understood strategies: “just enough” (ignoring unknown content) and adding schema extension points, which could be helpful for service evolution at technical level. Meanwhile, I. Robinson [39] discusses the “Consumer-Contract Driven” pattern, in which the changes is focused on the key business functionality demanded by consumers. In other words, those functionalities that don’t attract consumers are in no need of evolving, and thus could be thrown-out without side effect.

In [40], S. Wang proposes a quantitative analysis model based on inter- and intra-service dependency to
evaluate the impact of service evolution. Concentrating on human intention in service system evolution, [41] uses the situation framework to monitor, capture human intention and infer requirements for services. With this as the foundation, they propose a runtime service evolution mechanism, including releasing, feedback and packaging, to guide successor service development.

D. Requirements Evolution

It is commonly held that good understanding and control of requirements are of particular importance to the success of software. Many efforts have been made to improve the methods and tools in eliciting, specifying, and managing software requirements. On the topic of requirements evolution, requirements at first are roughly classified into two categories: stable and changing, and the later one have been further refined into mutable, emergent, consequential, adaptive, and migration requirements [42]. K. E. Wiegers [43] discusses four common tools for requirements management, which typically store requirements as objects or relations, and allow various operations, such as managing requirements versions/changes and linking requirements to test suites or design documents. N. Desai et al. [10] propose a methodology (named Amoeba) for modeling and evolving cross-organizational business processes based on protocols. Amoeba includes guidelines for handling the evolution of requirements (specified using protocols manually by designer) via a novel application of protocol composition. Moreover, N. A. Ernst et al. [44] exploit in detail about the goal-oriented requirements evolution problem, focus on finding a new specification for the changed requirements, on the basis of an inference engine - Requirements Engineering Knowledge Base (REKB). In his approach, he first checks to see whether the existing specifications are sufficient, if not, and then considers adding new specifications into the knowledge base.

III. CLASSIFIED DISCUSSION AND DISCOVERIES

In order to better understand software service evolution and identify principal research challenges, we are going to analyze evolution from four different dimensions, and for each one we will introduce its open problems and inadequacies through fitting literatures into our taxonomy.

A. Concept: Software Evolution vs. Service Evolution

Essentially, service is a particular piece of software. However, it differs from traditional software at many aspects, including development, delivery and consumption. For software evolution, the changes are closely related to code, module and architecture. In general, the studies on it fall into the following two classes: (a) analyzing the evolution trend of a software system in a long time span; (b) developing effective assets, techniques and approaches to support software evolution [1]. That’s to say, for software, evolution is regarded as both noun and verb. In comparing with software that is relatively stable and has longer update cycle, service is built to change and designed for responding quickly to ever-changing user requirements. Due to its intrinsic evolvability, the majority of work treats service evolution as a verb other than a noun and tries to develop theories and techniques to address those issues in evolving services.

Another important distinction is that service involves both consumer and provider. Service is designed for addressing the interoperability in application integration, thus a big challenge here is how to keep the compatibility between existing clients and evolved service. The incompatibility would arise when service change its interface or protocol. If look a little more deeply, we can discover that the change of service is driven by two kinds of requirements: external (from clients, e.g. new functionality, growing quality needs) and internal (from providers, e.g. reducing cost, using new techniques). The problem is that most of the studies focus on synchronizing clients with service and pay little attention to requirements, no matter consumers’ or providers’, that drive service evolution.

B. Abstraction: Macro vs. Micro

As stated in the previous section, service evolution can be considered at two abstraction levels, which are linked to the macro and micro perspectives respectively. From the macro viewpoint, services in an network is like individual organisms of a biological population, and Genetic Algorithms is applied to simulate the natural selection based on their fitness [17][18]; from the micro perspective, interfaces and interaction protocol to a service is similar with physical and behavior traits to an organism, many techniques and tactics have been exploited to keep clients compatible with service.

Closely related to macro perspective on service evolution is process evolution [45][46][47]. S. Rinderle et al. [45] present a Dynamic Choreographies (DYCHOR) framework which enables evaluating the change propagation of a process in a choreography on the basis of an extended automata model. In [46][47], the authors introduce business process evolution, of which the challenge is to dynamically migrate ongoing instances into the new version of a process.

On the topic of evolution, we cannot fail to mention adaptation, the counter part of it. Adaptation is also a very important mechanism to keep software and service behaving as expected under certain conditions. In service engineering, adaptation usually refers to (re-)select candidate services when suffering from failure [48][49], (re-)composite services for changed business process [50][51] and distribute new service instance in adapting to emergent work load [52]. The major concerns in service adaptation includes service composition, equivalence and substitutability [53][54].

The principal difference between adaptation and evolution is that adaptation won’t change (add/remove) service itself but evolution will. In other words, adaptation can only handle known situations, and when unknown environmental changes occur or requirements evolve, evolution - a procession of changes - would be essential.

C. Concern: Specification vs. Requirements

To the best of our knowledge, the-state-of-art of research on service evolution relies heavily on WSDL specification as well as FSM model. Both of these two groups commit themselves to the same problem - incompatibility between clients and evolved service, at different levels - interface and protocol. Though it is commonly held that the change of requirements is the key factor leading to service evolution, it still has not received enough recognitions.
Currently, there is some work studying requirements evolution based on Jackson & Zave’s definition of requirement problem [55]. The key concern is to find a new specification when requirements change, so that the changed requirements can still be satisfied. However, in service evolution, very few studies pay attention to requirements model, which could help us to identify what should evolve in a service. As it is, they usually assume a specified change pattern, and then exploit different tactics to resolve the subsequent incompatibility. At the same time, even for the incompatibility issue, they also fail to consider another important aspect: the cost of evolution, which is going to be impacted by both clients and service.

D. Methodology: Scientific vs. Engineering

Another inadequacy of service evolution is the lacking of theory and model, which constitutes its scientific foundation. As we can see from the literature review, the major approaches for tackling service evolution are engineering tactics, such as versioning, adaptor and design pattern. Until recently, V. Andrikopoulos et al. [28] has proposed a type and set theory based approach to reason whether certain changes to a WSDL service are allowed (i.e. compatible). The problem is that it also fails to model the root cause of evolution – requirements.

Along the main line from requirements to specification and then services, evolution is usually considered in isolation in each part and has not been connected together. Therefore, it is in need to develop theory, model, and techniques to study the change propagation from requirements to service as well as its inverse process.

E. Discoveries and Discussion

Through the extensive survey and detailed analysis, we have identified the following disciplines:

(1) Evolution is treated as both noun and verb in software area, but mainly as verb in service field.

(2) Software evolution is closely related to the change of code, module and architecture in tradition, and its latest work is on requirements evolution.

(3) Service evolution focuses on the compatibility between clients and evolved services at structural and behavior level, and pays little attention to requirements (both consumer and provider’s).

(4) The majority work on service evolution takes an engineering viewpoint, and scientific foundation for studying its evolution mechanisms is lacking.

Based on our analysis and discovery, it is clear to see that requirements are not treated in priority in current service evolution research. In fact, requirements could not only play a key role in driving service evolution but also (the non-functional ones) serve as the assessment criteria in selecting or recommending services. In our opinion, requirements driven service evolution is going to be an essential and promising research direction, especially in the user and service oriented context.

IV. CONCLUSIONS

Software services are subject to constant changes due to the change of requirements and environment. And the continuous changes would give rise to service evolution, which is of great importance in satisfying consumers’ varying requirements.

In this paper, we summarized the state-of-the-art studies on service evolution, including concepts, types, challenges and approaches. Based on the survey, we analyzed service evolution from four different dimensions, and for each one we discussed its research issues and inadequacies. Through the investigation and analysis, we find the leading driving factor for service evolution - requirements - is missing from current research and present requirements driven service evolution to be an essential and promising direction.

ACKNOWLEDGMENT

This research is funded by the ERC advanced grant 267856 "Lucretius: Foundations for Software Evolution", unfolding during the period of April 2011 - March 2016. Meanwhile, it is also supported financially by the National Basic Research Program of China (973) (Grant No. 2009CB320706)

REFERENCES


