Automatic Determination of Compatibility in Evolving Services

Karin Becker, Universidade Federal do Rio Grande do Sul, Brazil
Jim Pruyn, HP Laboratories, USA
Sharad Singhal, HP Laboratories, USA
Andre Lopes, HP Brazil R&D, Brazil
Dejan Milojicic, HP Laboratories, USA

ABSTRACT

A major advantage of Service-Oriented Architectures (SOA) is composition and coordination of loosely coupled services. Because the development lifecycles of services and clients are de-coupled, multiple service versions must be maintained to support older clients. Typically versions are managed within the SOA by updating service descriptions using conventions on version numbers and namespaces. In all cases, the compatibility among services descriptions must be evaluated, which can be hard, error-prone and costly if performed manually, particularly for complex descriptions. In this paper, the authors describe a method to automatically determine when two service descriptions are backward compatible. The authors describe a case study to illustrate version compatibility information in a SOA environment and present initial performance overheads. By automatically exploring compatibility information, a) service developers can assess the impact of proposed changes; b) proper versioning requirements can be put in client implementations guaranteeing that incompatibilities will not occur during run-time; and c) messages exchanged in the SOA can be validated to ensure that only expected messages or compatible ones are exchanged.

Keywords: Compatibility, Service Compatibility, Service Composition, Service Oriented Architecture (SOA), SOA Environment

INTRODUCTION

A major advantage claimed for service-oriented architectures (SOA) is the ease for making changes, because they support composition and coordination of autonomous, sharable services in a loosely coupled manner. SOA enables independent development by disparate teams, each one with its own delivery and maintenance schedule (Frank, 2008). The decoupled life-cycles of services and clients have major consequences from a change management perspective. As a service is upgraded, it must
continue to support existing clients while meeting requirements of new ones. This requires the ability to represent and manage multiple versions of the same service within the SOA, and transparently enable redirection of old clients to new versions of the service when possible. Ideally, compatible changes should not cause failures or unexpected behavior. Hence, research is required in how changes are introduced in services within the SOA (Andrikopoulous et al., 2008; Endrie et al., 2006; Fang et al., 2007a; Frank et al., 2007; Kaminsky et al., 2006). Finally, because every service can be used in multiple solutions, any change in the behavior of a service can cascade across clients (and clients of those clients) in a transitive manner, causing a broad impact within the SOA. Thus the SOA must support early detection of incompatible changes.

This paper addresses backward compatibility, which is concerned with how changes to the service interface affect existing clients (Andrikopoulous et al., 2008; Brown & Michael, 2004; Endrie et al., 2006; Fang et al., 2007b; Kaminsky et al., 2006; Narayan & Singh, 2007). In this paper, we shall use compatibility as synonymous with backward compatibility. A service version is defined to be backward compatible with a previous one if it a) delivers at least the same functionality; b) possibly relaxes constraints on the input expected while delivering the same results; and c) generates outputs that can be consumed by existing clients.

There is no comprehensive solution for managing compatibility of evolving services in SOA. Existing work can be divided into: a) best practices and design patterns for service versioning (e.g. Andrikopoulous et al., 2008; Brown & Michael, 2004; Endrie et al., 2006; Lubinsky, 2007; Narayan & Singh, 2007), b) version aware registry solutions (e.g. UDDI v3.0.2, Systinet, Fang et al., 2007b), which make assumptions about service compatibility, and c) architectural components for dealing with service compatibility within the SOA (e.g. Frank et al., 2007; Kaminsky et al., 2006). In all cases, it is assumed that compatibility among versions is assessed manually, a particularly error-prone task for complex service descriptions.

In this paper, we describe an approach for automatically assessing service compatibility between related versions of a service. Unlike related approaches, we do not infer compatibility from version number conventions, nor do we restrict changes between service versions to avoid incompatibility. The compatibility assessment method proposed is based on a version framework that allows service descriptions to evolve at different granularity levels, by considering a loose-dependency between the services and the elements used to describe them. We describe how the version framework and compatibility assessment method were prototyped in a SOA environment, and the lessons learned.

**SERVICE VERSIONING**

As pointed out by Frank et al. (2007), versioning is an overloaded term in the service context. From the client point of view, a service version refers to the “contract” established by the interface of the service, and consequently, to the functionality a service delivers and the data types which it exposes within the interface. From a service provider point of view, however, version refers to a particular implementation of the service, and how the service implementation evolves over time, regardless of changes in interface definition. We refer to these two aspects as the service model and service implementation, respectively. This paper concentrates on the evolution of the service model because it affects clients, and assumes that service implementation issues are handled using regular source versioning systems (Conradi & Westfechtel, 1998).

The service model is represented within the SOA by a model schema, which defines the external representation of the service as a set of versioned abstractions and relationships between those abstractions. We refer to these abstractions collectively as types. For instance, the model of a temperature forecast service is defined by a type describing the service itself, as well as other types such as Temperature, City,
Modern Diffusion of Products with Complex Network Models
[www.igi-global.com/chapter/modern-diffusion-products-complex-network/53279?camid=4v1a](www.igi-global.com/chapter/modern-diffusion-products-complex-network/53279?camid=4v1a)

[www.igi-global.com/chapter/sustainable-cloud-computing-frameworks-for-e-government/217836?camid=4v1a](www.igi-global.com/chapter/sustainable-cloud-computing-frameworks-for-e-government/217836?camid=4v1a)