Computer Languages, Systems & Structures 37 (2011) 2–23

Contents lists available at ScienceDirect

Computer Languages, Systems & Structures

journal homepage: www.elsevier.com/locate/cl

QoS-enabled and self-adaptive connectors for Web Services composition and coordination

J.L. Pastrana [a,*], E. Pimentel [a], M. Katrib [b]

[a] Departamento de Lenguajes y Ciencias de la Computacion, University of Malaga, Malaga, Spain
[b] Departamento de Computacion, University of Havana, Havana, Cuba

ARTICLE INFO

Article history:
Received 23 November 2009
Received in revised form
16 July 2010
Accepted 29 July 2010

Keywords:
Web Services
Composition
Coordination
Run-time adaptation
Ontology

ABSTRACT

This paper presents a methodology for Web Service composition and coordination based on connectors which are defined by Web Service client and automatically generated by the COMPOSITOR tool we have developed. Connectors use contracts to express the non-functional requirements and the behaviour desired by the client of a service, such as QoS (Quality of Service) features. The connectors generated are self-adaptive. The adaptation enactment is based on using an OWL ontology of the server domain which makes it possible to adapt any mismatch in a call to a service at run-time when the server is updated or replaced.

1. Introduction

A Web Service is a collection of functions packaged together and published on a network to be used by client programs. It is defined using common, Web-related technologies: TCP/IP, HTTP, FTP, SMTP, and XML as well as three specifications: Simple Object Access Protocol (SOAP), Web Services Description Language (WSDL) and Universal Description Discovery and Integration (UDDI) which together form set of technologies that define Web Services.

A Web Service receives, accepts and processes SOAP-encoded messages from clients. If the client and the server applications were developed by the same team, then little more would be needed to make this happen; the client team would have all the information it needed to move ahead. However these applications are typically developed not only by different teams, but also often by different organizations or parts of an organization.

Depending upon which part of the system is being developed by a team, the understanding of the other system is limited to the protocol or specification that defines the Web Service. A WSDL document is an XML formatted document that is like an interface definition for a class. It describes how a client can connect to, and invoke, a service.

The application developer is responsible for discovering a suitable Web Service. And the Universal Description Discovery and Integration (UDDI) specification provides the necessary support by defining a standard mechanism for publishing and locating businesses and the services they provide. Typically, client developers determine during the application design which Web Services and providers the application will use.

* Corresponding author. Tel.: +34 95 213 3316; fax: +34 95 213 1397.
E-mail addresses: pastrana@lcc.uma.es (J.L. Pastrana), ernesto@lcc.uma.es (E. Pimentel), mkm@matcom.uh.cu (M. Katrib).
URLS: http://www.lcc.uma.es/~pastrana (J.L. Pastrana), http://www.lcc.uma.es/~ernesto (E. Pimentel), http://www.weboo.matcom.uh.cu (M. Katrib).

1477-8424/$ - see front matter © 2010 Elsevier Ltd. All rights reserved.
doi:10.1016/j.cl.2010.07.001

Dynamic software architectures modify their architecture and carry out the modifications during the execution of the system. This behaviour is known as run-time evolution. Self-adapting architectures are a specific type of dynamic software architecture. Systems which can monitor and adapt to the changes in their environment are known as self-adaptive, self-healing, or self-managing systems.

Self-adaptive software systems [7,9] are those able to manage changing operating conditions dynamically and autonomously. Such self-adaptive systems can configure and reconfigure themselves, augment their functionality, continually optimize themselves, protect themselves, and recover themselves, while keeping most of their complexity hidden from the user and administrator. Currently, most proposals in this field rely on an explicit representation of the components and goals of the system (usually following a top-down approach), or on the definition of local rules for the different elements of the system, which results in an emergent self-organizing behaviour (hence following a bottom-up approach). Both these approaches are suitable for closed systems, that is, those whose constituent components are well known at design time, or where there is no need to explicitly represent the goals of the system. However, there are many situations in which software systems are open, and the changes in their execution environment are directly subject to the availability of a particular service, which may join or leave the context of the system at any given moment. These systems lack a predefined description of their architecture and components, and even of their goals. In such open systems, new adaptability problems arise, such as the connection and disconnection of a new software component to an already running system; or how to solve interoperability issues among third-party components not specifically designed to interact with each other.

The case study of a library application where users can load/upload papers has been selected, in order to explain the different concepts with a consistent reference scenario that runs through the whole paper. Basically, the application will use an external Web Service for papers storage and management and both work together in a loosely coupled way. This case study will be used to show how using our framework, the non-functional and QoS requirements of a client can be expressed using Meyer design by contract style (for example, suspending a call to synchronize, ignoring a request and returning a new value, null object refactoring, changing the server used when the average responding time is greater than wanted, etc.), and how the framework can automatically solve the run-time adaptation problems caused when the server is updated or replaced by other semantically equivalent.

1.1. State of the art

The literature on Web Services and the Semantic Web is abundant [29] and the need for a more rigorous formal foundation is widely discussed. Currently, most of the existing work is about Web Services description, the syntax flow and their execution.

There are several standardization efforts to extend the expressiveness of the Web Service Definition Language (WSDL) with support for defining extra-functional properties. For example, the WS-Policy standard [4] can be used to specify policies expressing non-functional requirements for Web Services (WS), while [31] proposes a language, WS-QoS, for specifying provided and required Web Services QoS. Those proposals are very useful and interesting; however, they address expressing the non-functional properties of the server instead of the requirements of the clients. Furthermore, they do not consider the run-time adaptation problem.

There are many Web Services flow specification languages like WS-BPEL [13] and WSCI [1] to describe in which order messages have to be exchanged between services. The composition of the flow is still manually obtained. Semantic annotations have been widely discussed in the Semantic Web community where preconditions and the effects of services are explicitly declared in the Resource Description Format (RDF). WS-BPEL provides a programming-language like constructs (sequence, switch, while, pick) as well as graph-based links that represent additional ordering constraints on the constructs.

In [19], a method is presented to compose Web Services by applying logical inferencing techniques on predefined plan templates. The service capabilities are annotated in DAML-S/RDF and then manually translated into Prolog. Now, given a goal description, the logic programming language of Golog [16] (which is implemented over Prolog) is used to instantiate the appropriate plan for composing the Web Services. Golog is based on the situation calculus and it supports specification and execution of complex actions in dynamic systems. The Golog reasoner, given the plan and action templates, evaluates choices in a non-deterministic way and executes the plan. Since execution is in the Prolog environment, the non-deterministic choice is actually made according to the default evaluation order. Essentially, Golog programs are user provided plan templates which are customized to goal instances. The system uses hand-built wrappers to transform semantic annotations into Golog representations, and vice versa.

However, BPEL and Golog do not address compliance and QoS, cannot adapt service calls at run-time, and they deal with connectivity only, not with correctness, furthermore, they cannot verify properties of a composition result.

The ASTRO approach [18] works on the idea of the synthesis of a Web Service composition consists in understanding how to orchestrate the interactions among the component services, so that their protocols are respected, all non-deterministic outcomes are covered, and the composition requirement is achieved. The ASTRO proposal presents a formal framework for the automated composition of Web Services that is able to cope with Web Services exposing complex