Evolução Controlada de Arquitecturas de Serviços Web

João Campinhos, João Costa Seco, and Jácome Cunha

NOVA LINCS**, DI, FCT, Universidade NOVA de Lisboa

As arquitecturas de aplicações baseadas em micro serviços, ou aplicações baseadas em serviços web, têm tido um crescimento significativo, muito pelo exponencial aumento da utilização de dispositivos móveis, cujas aplicações dependem quase totalmente deste tipo de interfaces. A evolução destes serviços constitui um problema de engenharia de software com impacto em, potencialmente, muitos milhões de utilizadores. Qualquer modificação ao comportamento ou à estrutura de uma interface remota pode causar erros de execução, uma vez que a sua actualização depende das componentes ou serviços [3]. De facto, os programadores podem até desconhecer a versão exacta das diferentes componentes que usam, tornando a sua utilização propícia a erros [2].

Para resolver este problema, introduzimos neste trabalho um modelo de programação que permite uma evolução segura de uma interface de serviços web, minimizando a introdução de incompatibilidades que poderiam eventualmente gerar um comportamento inesperado. A abordagem apresentada passa por permitir que uma mesma interface de programação disponibilize simultaneamente múltiplas variantes que correspondem a várias versões tornadas públicas no desenvolvimento. O nosso principal objectivo é o de permitir uma fácil evolução do código fonte, garantindo que as aplicações cliente do sistema, de versões anteriores, são correctamente suportadas, e não são descontinuadas abruptamente por erros de execução. Usamos técnicas de construção de linguagens de programação para garantir que o versionamento é tratado de forma modular, sendo que a nossa abordagem apenas exige que o programador definia uma relação de ordem entre versões do seu sistema de versionamento. O sistema de verificação (estática) de compatibilidade de código entre versões (tipificação) usa a relação de ordem definida para aferir a correcção das modificações introduzidas.

Exemplo Para ilustrar o processo de evolução de software que o sistema suporta, considera-se o exemplo seguinte, onde se define um sistema servidor em que um dos métodos (`bar`) é definido duas vezes, em duas versões distintas. Também se define um cliente, que executa numa versão explicitamente indicada. Utilizamos nesta ilustração um sistema de versões do tipo *Major.Minor* (e.g. `@0.1`, `@2.0`), em que a relação de ordem é a ordem natural dos números das versões. Neste exemplo permitimos alterações das assinaturas dos métodos entre versões *Major*.

```
Server = {foo()@0.1 = "foo",          Client = {@3(foo();
          bar()@1.0 = "bar",                    bar())}
          bar()@2.0 = true}
```

** Trabalho apoiado por NOVA LINCS ref. UID/CEC/04516/2013.

Na linguagem modelo que usamos, os métodos são explicitamente anotados (`foo()@0.1`). Introduz-se também uma operação de mudança explícita de contexto (`@3...`), que executa uma mudança da versão em que as expressões são avaliadas. Para o código do cliente apresentado, o servidor devolverá a versão *Minor* mais recente cuja *Major* é 3. Não existindo nenhuma versão nessas condições (como é o caso), o nosso sistema executa um algoritmo que utiliza a relação de ordem especificada, para procurar a versão mais próxima da 3. Neste caso, as versões devolvidas ao cliente são `foo()@0.1` e `bar()@2.0`, pelo que o resultado do cliente será `"foo"` e `true`, respectivamente.

Tipificação A relação entre versões é utilizada pelos nossos algoritmos de procura de versões e de análise estática de tipos para aferir a correcção das alterações introduzidas entre versões. No exemplo que usamos referimos que entre versões *Major* era possível modificar a assinatura dos métodos, tornando assim o servidor válido. No entanto, a adição do método `bar()@2.1 = 4` ao servidor faz com que o sistema emita um erro de tipos, uma vez que a relação de versões não permite mudar o tipo de retorno da versão 2.0 para a versão 2.1 (versões *Minor*).

Implementação De forma a validar a nossa abordagem, estamos a implementar uma extensão da linguagem JavaScript (JS). Para anotar o código do servidor com versões o programador deve usar a sintaxe do Flow [1] – um analisador estático de tipos para JS que recorre a anotações. A nossa implementação transforma a AST original removendo as anotações das versões e fazendo as necessárias alterações ao código para que este esteja correcto (e.g. renomeação). O Flow é ainda usado para garantir que o código final está correcto. A nossa implementação estende ainda a plataforma *expressjs* de forma a suportar a especificação do contexto de versão no cabeçalho do pedido do cliente.

Conclusão Nesta comunicação apresentamos uma técnica para o desenvolvimento e evolução controlada de serviços web, bem como dos seus clientes, através da introdução de um sistema de gestão de versões de código fonte. A execução e mudança de versões é assegurada pelo sistema que executa os pedidos do cliente consoante o contexto desejado. Assim, correcções e alterações ao código do servidor são reflectidas no cliente sem esforço do seu programador. Este sistema garante a execução de múltiplas versões do sistema, ao mesmo tempo que promove uma prática mais responsável de evolução de serviços web. Por ser um sistema modular e genérico, é facilmente adaptável para múltiplos sistemas de versionamento, bastando para isso definir uma relação de ordem entre as versões.

Referências

1. A. Chaudhuri. Flow: a static type checker for JavaScript. SPLASH-I In Systems, Programming, Languages and Applications: Software for Humanity, 2015.
2. M. W. Godfrey and D. M. German. The past, present, and future of software evolution. In *Frontiers of Software Maintenance (FoSM'08)*, pages 129–138, 2008.
3. J. M. Gonzalez-Barahona, G. Robles, M. Michlmayr, J. J. Amor, and D. M. German. Macro-level software evolution: A case study of a large software compilation. *Empirical Software Engineering*, 14(3):262–285, 2009.