Substitución de Componentes Software basado en Testing

Andrés Flores

Grupo GIISCo
Departamento de Ciencias de la Computación
Universidad Nacional del Comahue
Neuquen, Argentina
Tel. +54 299 4490300 ext. 433
Fax +54 299 4490313
Email: aflores@uncoma.edu.ar
Web: http://giisco.uncoma.edu.ar

Macario Polo

Grupo Alarcos
Escuela Superior de Informática
Universidad de Castilla-La Mancha
Ciudad Real, España
Tel. +34 926 295300 ext.3747
Fax +34 926 295354
Email: macario.polo@uclm.es
Web: http://alarcos.inf-cr.uclm.es

Resumen
Los componentes software permiten la crear aplicaciones a través de ensamblaje, y realizar ajustes y extensión al agregar o cambiar un conjunto de componentes interdependientes. Sin embargo, esta flexibilidad conlleva riesgos sino se administra cuidadosamente, dado que de otra manera se podría incurrir en la introducción faltas (faults) o pérdida de funcionalidad debido a componentes incompatibles. Así, resulta imperativo la verificación de la posibilidad de substitución de componentes software. Por esto estamos desarrollando un procedimiento de evaluación a nivel sintáctico y semántico, donde este último se basa en la aplicación de estrategias de testing y usando herramientas de testing de caja negra, como son JUnit y NUnit para el caso de componentes Java o .Net respectivamente.

Area: Ingeniería de Software y Base de Datos

Palabras Clave: Ingeniería de Software basada en Componentes, Testing de Componentes Software.

1. Introducción

Los componentes software permiten que los ingenieros puedan crear aplicaciones a través de procedimientos de ensamblaje, y además realizar ajustes y extender la funcionalidad simplemente agregando o cambiando un conjunto requerido de funcionalidades en la forma de componentes interdependientes. Esta es la razón por la cual existe una gran cantidad de modelos y frameworks de componentes.
Desde el punto de vista de los desarrolladores, la corrección de errores y la actualización resulta más fácil en aplicaciones basadas en componentes que en arquitecturas monolíticas, dado que sólo deben ser reemplazados los componentes afectados y no la aplicación completa.
Sin embargo, esta flexibilidad tiene un cierto precio – el reemplazo (ya sea una nueva versión o una substitución más general) debe ser cuidadosamente administrado, porque de otra manera se podría incurrir en la introducción de un componente incompatible. Las posibles consecuencias de una substitución tal pueden involucrar la inyección de una falta (*fault*) que genere un comportamiento peculiar no inmediatamente distinguible, o bien directamente una pérdida de funcionalidad de la aplicación como un todo.

Es por ello que resulta imperativo el apropiado estudio de técnicas y herramientas para verificar componentes de reemplazo ante la necesidad de substitución de componentes. A pesar de esto, las soluciones en la industria se basan principalmente en procesos ad-hoc y mayormente manuales [11].
Tanto para arquitecturas software aisladas como para aquellas que conviven e interactúan en alguna infraestructura integradora subyacente, se requiere contar con un Modelo de Aplicaciones que provea la especificación de la arquitectura describiendo los elementos que la conforman, es decir los Modelos de Componentes involucrados. Un Modelo de Componente provee una definición para instanciar un componente junto con los aspectos de composición, los cuales describen interacciones estándar e interfaces no ambiguas [5,8,9].
Para asegurar que un componente de reemplazo dado es el adecuado con respecto a un Modelo de Aplicación, se necesita por lo tanto realizar una evaluación de su Modelo de Componente. Para ello presentamos un procedimiento de evaluación que toma aspectos funcionales de los componentes y los compara con la especificación provista por el Modelo de Aplicación.
El primer paso en la evaluación es analizar los servicios del componente a nivel sintáctico, que permite descubrir casos de incompatibilidad y por lo tanto ayuda a reducir el tiempo global del proceso de evaluación [6,3]. El segundo paso se enfoca en el comportamiento del componente a un nivel semántico. Para esto analizamos los datos de entrada y salida, y cómo se produce la transformación de datos – es decir, la conversión de datos de dominio en datos del rango [1] – lo cual ayuda a deducir la funcionalidad interna que oculta el componente. El procedimiento está basado en la aplicación de estrategias de testing por medio del uso de herramientas de testing de caja negra ampliamente conocidas.
En un trabajo previo [7] hemos desarrollado un Meta-Modelo cuyo propósito es la generación de Casos de Test, los cuales son luego ejercitados por medio de las herramientas JUnit o NUnit para el caso de componentes Java o .Net respectivamente. En esta instancia, realizamos la aplicación de ese modelo en la evaluación semántica de componentes, considerando que los casos de test pueden ser generados en función de una versión Java o .Net del Modelo de Componente requerido. El enfoque se ajusta al aspecto de *facilidad de observación* que se corresponde a la cualidad de facilidad de testing. La facilidad de observación trata con la posibilidad de observar el comportamiento de un componente de acuerdo a su comportamiento operacional y a sus salidas en función de sus entradas [4,2].
El análisis sintáctico no solamente verifica una compatibilidad de interfaz, sino también permite obtener una colección de posibles correspondencias entre servicios del componente de reemplazo y aquel que debe ser substituido (componente de referencia). Tal correspondencia de servicios será usada para generar drivers que ejecuten casos de test que hubieren sido generados para el componente de referencia (a ser substituido). De esta manera estamos en presencia de un modelo de análisis mutacional o en particular de perturbación de la interfaz. Tal análisis podría ser incluso mejorado por medio de una perturbación de los parámetros incluidos en la signatura de los servicios.
Al efectuar la ejecución de los casos de test sobre los drivers generados se podrían obtener tres conjuntos de acuerdo a los resultados obtenidos: (1) drivers con todos los resultados fallidos, (2) drivers con una cierta cantidad de resultados exitosos, y (3) drivers con todos los resultados exitosos. El primer caso tiene el beneficio de descartar tales drivers o mutantes, el segundo caso debe ser analizado de acuerdo al porcentaje de éxito, y el tercer caso se esperaría que el conjunto tuviera un únicamente miembro, para así obtener la correspondencia esperada entre los servicios del componente de referencia y el de reemplazo. De esta manera luego se puede fácilmente generar un wrapper adaptador para el componente de reemplazo, de manera que aquellos que interactuaban con el componente de referencia, ahora puedan hacerlo con el de reemplazo con mínimos o ningún cambio.
En esta instancia nos encontramos desarrollando funcionalidad para mejorar la evaluación sintáctica y permitir la generación de los drivers con la permutación de la interfaz. Luego de ello, procederemos a

la experimentación para obtener resultados que nos permitan establecer adecuados umbrales de compatibilidad.

2. Agradecimientos

Este trabajo está financiado por los siguientes proyectos: CyTED-CompetiSoft (506AC0287), UNCo-MPDSbC (04-E059), UCLM-CALIPSO (TIN2005-24055-E), y UCLM-ESFINGE (TIN2006-15175-C05-05)

3. Referencias

1. Alexander, R. and Blackburn, M. *Component assessment using specification-based analysis and testing*, Technical Report SPC-98095-CMC, Software Productivity Consortium, Herndon, Virginia, USA, 1999.
2. Cechich, A., Piattini, M. and Vallecillo, A. Component-based Software Quality: Methods and Techniques," LNCS 2693, Springer-Verlag, 2003.
3. Flores, A. and Polo, M. *An Approach for Application Suitability on Pervasive Enviroments*, in: *3rd IWUC'06, held during ICEIS'06*, Paphos, Cyprus, 2006, pp.71-78.
4. Freedman, R. S., *Testability of Software Components*, Transactions on Software Engineering 17, pp. 553-564, 1991.
5. Heineman, G. and Council, W. Component-Based Software Engineering - Putting the Pieces Together," Addison-Wesley, 2001.
6. Polo, M. and Flores, A. *Towards Run-time Component Integration on Ubiquitous Systems*, in: *3rd MSVVEIS'05, held during ICEIS'05*, Miami, Florida, USA, 2005, pp. 9-18.
7. Polo, M., Tendero, S. and Piattini, M. *Integrating Techniques and Tools for Testing Automation*, InterScience: Journal of Software Testing, Verification and Reliability 16, pp. 1-37, 2006. http://www.interscience.wiley.com.
8. Warboys, B., Snowdon, B., Greenwood, R., Seet, W., Robertson, I., Morrison, R., Balasubramaniam, D., Kirby, G. and Mickan, K. *An Acive-Architecture Approach to COTS Integration*, IEEE Software, pp. 20-27, 2005.
9. Szyperski, C. *Component Software Beyond Object Oriented Programming*, Addison-Wesley, 1998.
10. Chaki, S., Clarke, E., Sharygina, N., Sinha, N. *Verification of Evolving Software via Component Substitutability Analysis*. Technical Report CMU/SEI-2005-TR-008, Carnegie Mellon University December 2005.
11. Brada, P., Valenta, L. *Practical Verification of Component Substitutability Using Subtype Relation*, EUROMICRO-SEAA'06, 32nd Conference on Software Engineering and Advanced Applications, Dubrovnik, Croatia, August 29-September 1.