Verificación y validación mediante un modelo de componentes

Agustín Cernuda del Río
Jose Emilio Labra Gayo
Juan Manuel Cueva Lovelle
Universidad de Oviedo - Departamento de Informática
Facultad de Ciencias - C/ Calvo Sotelo, S/N
Oviedo, España, 33007
{guti, labra, cueva}@lsi.uniovi.es

RESUMEN

La interpretación abstracta, el análisis de flujo y otras técnicas de verificación y validación persiguen comprobar la corrección de un sistema software a priori (sin ejecutarlo). Como contrapartida, la aplicación de estas técnicas puede requerir una importante base matemática, y además es frecuente que su aplicabilidad esté limitada a ciertos niveles de abstracción.

La noción (abierta) de "componente software" puede jugar un papel importante en la verificación de la calidad; un modelo de componentes simple y versátil permite incorporar conocimiento al proceso de desarrollo, y la adecuación de los componentes entre sí puede verificarse incluso sin necesidad de construir el programa. Además, puede aplicarse en muy diversos niveles de abstracción de la producción de software, lo que presenta ventajas frente a las técnicas mencionadas.

Palabras clave: Componente software, verificación, validación, modelo de componentes, ingeniería del software, proceso de desarrollo del software.

ABSTRACT

Abstract interpretation, data flow analysis and other verification and validation techniques try to verify that a software system is correct without actually running it. The drawback is that applying these techniques may involve strong mathematical skills, and in addition it is usual that their applicability is limited to certain abstraction levels.

The (open) notion of "software component" can play an important role in quality verification; a simple, versatile component model allows the incorporation of knowledge to the development process, and component proper matching can be verified even without building the program. In addition, it can be applied at very many different abstraction levels of the development process, and this is an important advantage over other techniques mentioned above.

Keywords: Software component, verification, validation, component model, software engineering, software development process.

1 INTRODUCCIÓN

En los últimos años ha tenido un cierto impacto la noción de "componente software", que habitualmente [16] se asocia con la idea de un elemento ya compilado (librería, módulo o similar), generalmente desarrollado por terceras partes, y que se adquiere con el fin principal de ahorrar tiempo de desarrollo. Construyendo los sistemas mediante la combinación de estos componentes, se espera ahorrar tiempo y mejorar la calidad (se supone que los componentes están bien probados y validados por su uso generalizado).

No obstante, esto no siempre es así. Independientemente de la calidad de un componente individual, se ha demostrado que la combinación de varios componentes puede generar problemas nuevos de tipo *emergente* que no resultan de la mera adición de errores [1, 9, 10].

Además de esto, no suele hacerse hincapié en la necesidad de métodos que permitan verificar que realmente se cumplen todas las condiciones que cada fabricante ha asumido como nominales. Estos requisitos no suelen estar formalizados de manera que puedan verificarse automáticamente, sino que suelen residir en documentación destinada a lectores humanos. En general, los así denominados "modelos de componentes" de mayor aceptación comercial, tales como ActiveX, CORBA o JavaBeans, suelen servir como simple apoyo para resolver problemas de interoperabilidad entre plataformas o similares. Las precondiciones y postcondiciones, (si es que se utilizan) son porciones ejecutables que en cierto sentido plantean dificultades similares a las que plantean las pruebas.

En este artículo se evalúan técnicas de verificación cuyo fin es evaluar a priori la corrección de un sistema (sin necesidad de ponerlo en ejecución). A continuación, se presenta un modelo de componentes (de forma general y obviando los detalles por razones de extensión), desarrollado bajo restricciones de simplicidad y flexibilidad, que pretende abordar esa misma problemática incorporando conocimiento en la especificación de un componente, siendo aquí el concepto de "componente" deliberadamente flexible; posteriormente, se describen los prototipos desarrollados para evaluar la viabilidad práctica, y se presentan después diversos casos de aplicación de este modelo, que permiten entrever su potencial. Especialmente, se describe un sistema de diagnóstico y gestión de configuraciones a distancia basado en este modelo.

2 INTERPRETACIÓN ABSTRACTA Y ANÁLISIS ESTÁTICO

Es bien sabido que las –necesarias- actividades de prueba del sistema plantean serias dificultades para su realización. El potencial espacio de estados que un sistema software puede alcanzar es enorme, y a efectos prácticos suele ser imposible recorrerlo por completo. Por tanto, es necesario seleccionar un subconjunto significativo de dicho espacio de estados, basándose en criterios –en gran medida heurísticos- recogidos en la literatura sobre ingeniería del software.

No es reciente la idea de recorrer este espacio de estados de forma analítica. Al fin y al cabo, la resolución analítica de problemas se ha aplicado con éxito en multitud de áreas de la matemática. Sin embargo, la automatización de este análisis en el caso que nos ocupa resulta especialmente difícil; es bien sabido que el análisis de programas de ordenador mediante otros programas de ordenador presenta dificultades especiales, e incluso límites absolutos a la computabilidad.

La **interpretación abstracta** [6, 13]es una aproximación a este problema. Se basa en que la semántica de un lenguaje de programación puede ser más o menos precisa, según el nivel de observación elegido. Si se observa el programa sin exigir un alto nivel de detalle, se puede realizar una abstracción de su semántica que, aun siendo menos precisa que la original, en contrapartida es computable. El **análisis estático** se basa en la interpretación abstracta para, dada la semántica original de un programa, derivar otra semántica aproximada y computable. De este modo, los ordenadores pueden analizar en tiempo de compilación el comportamiento que un programa tendrá en tiempo de ejecución y prever problemas, lo que lógicamente resulta fundamental de cara a la calidad.

El grupo pionero en este campo de investigación es probablemente el encabezado por Patrick Cousot, cuya actividad comenzó ya en los años 1970 [4, 5, 6]. En estos trabajos se presentan algoritmos de análisis estático que, por ejemplo, permiten evaluar un programa y detectar relaciones lineales entre los valores de sus variables.

No obstante, aun habiéndose desarrollado bases teóricas para la interpretación abstracta hace ya más de veinte años, muy pocos proyectos incorporan estas ideas (buena parte de las aplicaciones prácticas de análisis estático están encaminadas a la optimización de código en compiladores, y no a la detección temprana de defectos en programas).

Hay otra reflexión que hacer a este respecto: en ocasiones, lo que se desea verificar en un sistema no es su comportamiento a un nivel algorítmico riguroso. Frecuentemente, los programadores o diseñadores tienen un conocimiento sobre el sistema que podría ser aprovechado sin un análisis del tipo descrito; es evidente que muchos

programadores son capaces de afirmar que un programa dado contiene un error o que se bloqueará, aunque no sean capaces de demostrarlo algorítmicamente. Si todo el conocimiento que programadores, diseñadores, fabricantes, etc. tienen sobre los componentes software pudiera describirse formalmente y de manera sencilla, sería posible generar una "base de conocimientos" que describe todo lo que se sabe sobre el sistema que se está construyendo; esto permitiría verificar si los requisitos de ciertos componentes se cumplen o no, y por tanto prever defectos de funcionamiento. Un sistema de inferencia convencional, basado en lógica de primer orden, podría realizar estas verificaciones con relativa facilidad.

3 EL MODELO DE COMPONENTES ITACIO

En esta línea, se ha desarrollado un modelo de componentes [2, 3, 12], denominado **Itacio**, que pretende incorporar las ideas esbozadas en el párrafo anterior. Este modelo se ciñe a unas restricciones básicas:

- **Simplicidad**, gracias a la cual se pretende conseguir objetivos como los siguientes:
  - **Fácil comprensión del modelo**. Se pretende que su aplicación práctica resulte atractiva, disminuyendo los costes de aprendizaje y posibilitando de hecho que se implante en el proceso de desarrollo. Esto no ha ocurrido con otros métodos formales.
  - **Flexibilidad**. Un modelo simple puede aplicarse a diversos propósitos y niveles de abstracción.
- **Viabilidad técnica**. Se pretende que el modelo se pueda implementar con tecnologías existentes y a un coste razonable, sorteando, por ejemplo, los problemas de computabilidad o decidibilidad mencionados. En nuestro caso, con recursos mínimos se han podido implementar prototipos funcionales, lo que mueve al optimismo respecto a su viabilidad.

Lógicamente, la noción central del modelo es la de *componente*. Como ya se ha adelantado, en Itacio un componente no es necesariamente un módulo ejecutable; es simplemente un artefacto software con fronteras definidas. Por tanto, pueden ser componentes las instrucciones individuales, los contratos de reutilización [14, 15], las clases/objetos, o muchos otros elementos software a diferentes niveles de abstracción.

La frontera de un componente es "permeable"; presenta puntos de conexión que pueden ser *fuentes* ("salidas" del componente) o *sumideros* ("entradas" en el componente). El componente contiene, además, *expresiones restrictivas* que hacen referencia a las fuentes y sumideros. Estas expresiones pueden ser *requisitos* (plantean exigencias sobre los sumideros) o *garantías* (afirmaciones respecto a las fuentes). La creación de un *sistema* exige la composición de varios de estos componentes de modo que una fuente de un componente se conecta con un sumidero de otro componente (esta relación no tiene por qué ser de 1 a 1, pero sí de fuente a sumidero en cualquier caso, es decir, no se admite la conexión entre fuentes o entre sumideros).

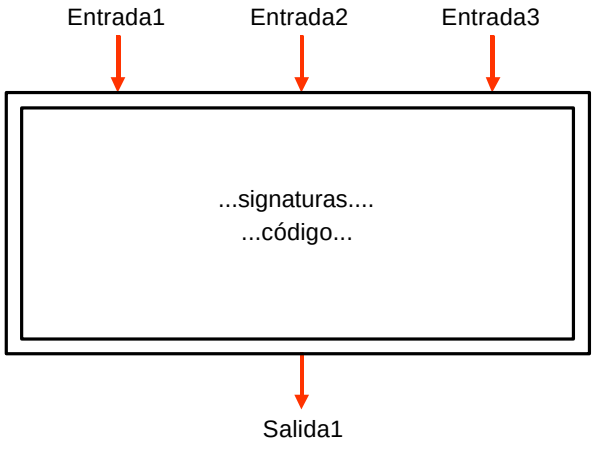

Fig. 1. Verificación tradicional de las conexiones de un componente.

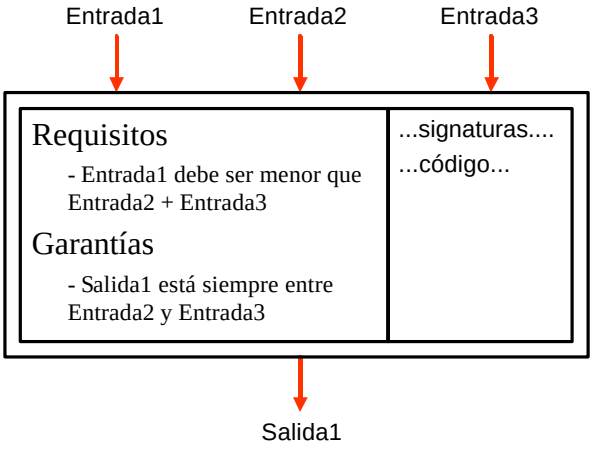

Fig. 2. Verificación en Itacio gracias a las expresiones restrictivas.

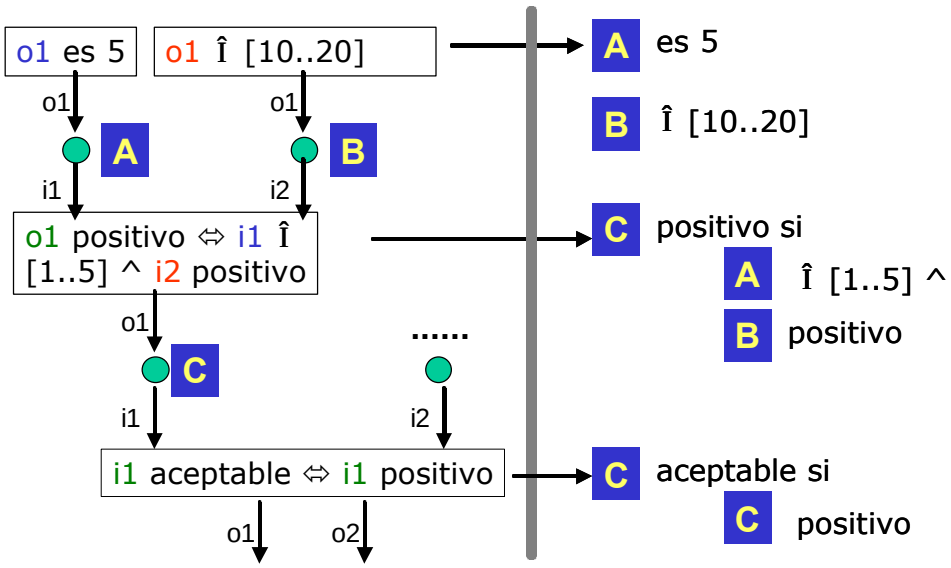

Fig. 3. Ejemplo de generación de la Base de Conocimientos en Itacio. A la izquierda, estructura del sistema de componentes. A la derecha, base de conocimientos que lo describe. Mediante inferencia, es posible saber si la conexión C es aceptable o no.

Dado uno de estos sistemas, el proceso de verificación comienza con la generación de una *base de conocimientos* que lo describa (Fig. 3). En Itacio, las expresiones restrictivas se denotan utilizando lógica de primer orden en forma de cláusulas Horn, y se implementan sobre un lenguaje de Programación Lógica con Restricciones (CLP, de *Constraint Logic Programming*) [8]. La base de conocimientos se genera reuniendo todas las expresiones –requisitos y garantías- de todos los componentes; si una fuente y un sumidero están conectados, se realiza una sustitución de modo que todas las apariciones de esa fuente o ese sumidero hagan referencia a un mismo identificador creado al efecto. Así, la información sobre la interconexión de los componentes queda implícitamente recogida.

Creada la base de conocimientos, sólo resta verificar que todas las exigencias de todos los componentes se cumplen; un sistema de inferencia convencional puede realizar deducciones sobre esta base de conocimientos con garantías de compleción. Si una restricción no se satisface, el sistema de inferencias puede indicar exactamente qué es lo que falla, dónde, y dar explicaciones acerca del problema.

4 PROTOTIPOS

Ya se ha mencionado que la noción de "componente" que se plantea aquí es algo deliberadamente abierto, y por tanto flexible. Se trata de producir un modelo genérico que se pueda aplicar de muy diversas formas.

Para comprobar la viabilidad de este enfoque, se han probado diferentes aplicaciones del modelo. Para estos experimentos resulta imprescindible disponer de soporte en las siguientes áreas funcionales:

- Un motor de inferencias para lógica de primer orden.
- Editores para describir los componentes y los sistemas.
- Sistemas de representación para mostrar los resultados de la validación.

El **motor de inferencias** puede implementarse sobre elementos ya existentes; en este caso, ha bastado con recurrir a un sistema de Programación Lógica con Restricciones (CLP), llamado ECLiPSe [7, 17]. Esto ha permitido comprobar hasta qué punto era acertado nuestro enfoque; recuérdese que una de las premisas del proyecto era la viabilidad técnica. Los sistemas de inferencia llevan ya muchos años siendo utilizados, y los lenguajes de programación declarativa constituyen un paradigma de programación ya conocido y que no resulta extraño a muchos programadores. La mera integración de ECLiPSe en los diversos prototipos ha permitido solventar con suma facilidad y a bajo coste uno de los problemas básicos del modelo.

Por lo que se refiere al **subsistema de edición**, un primer prototipo se basaba en SEDA, una herramienta gráfica

extensible propiedad de Seresco, S.A.; posteriormente, para facilitar la instalación y soporte (y dado que el prototipo no requería facilidad de uso) se optó por simples ficheros de texto. Una serie de clases desarrolladas en Java tratan estas descripciones y generan la base de conocimientos correspondiente al sistema evaluado. La interfaz de usuario es de tipo web, con el fin de ahorrar esfuerzo de desarrollo.

Finalmente, el **subsistema de representación** también ha pasado por varias etapas; inicialmente, los resultados se presentaban también en SEDA, pero al eliminar la dependencia de esta herramienta se optó por recurrir a un modelo de visualización web, basado en XML/XSL/VML.

El prototipo resultante no es, por supuesto, una herramienta que se pueda utilizar en producción; pero la enseñanza que se obtiene de esto es que al fin y al cabo los problemas básicos de implementación del sistema se han podido solventar con recursos humanos muy limitados y con poca exigencia de recursos técnicos. No ha sido necesario en absoluto forzar el "estado del arte" en el plano técnico, cosa que sí pueden requerir otras técnicas de interpretación abstracta.

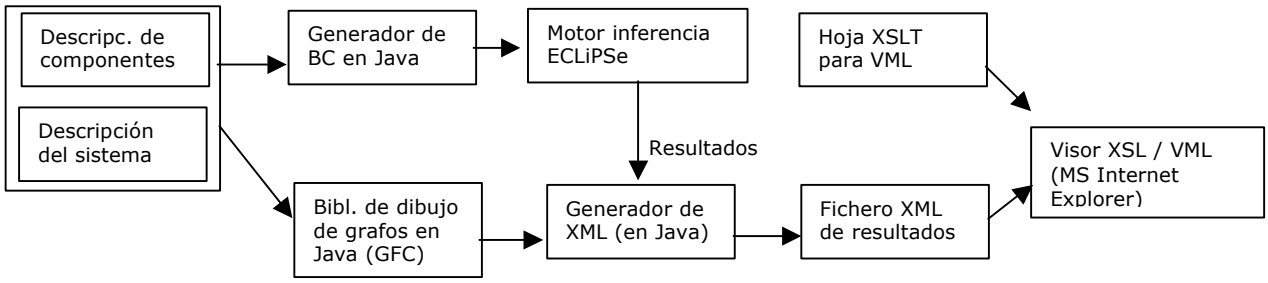

Fig. 4. Arquitectura del prototipo para Itacio.

5 CASOS DE APLICACIÓN

El prototipo descrito en el apartado anterior constituiría el sistema de descripción y verificación de sistemas (genérico); el paso siguiente es comprobar que puede aplicarse a diferentes tipos de problemas, incluyendo los que habitualmente no se vinculan al campo de los componentes. Basta con realizar un proceso de *instanciación* del modelo, redefiniendo en cada caso qué se entiende por componente, flujo y sumidero; hecho esto, sólo resta escribir las especificaciones de los componentes y verificar el sistema.

Microcomponentes

Un primer campo de aplicación es el de los microcomponentes. Se trata de considerar que el código –a nivel de lenguaje de programación- está compuesto por componentes. Aunque los elementos de un lenguaje de programación pueden combinarse de infinidad de maneras, lo cierto es que un programador experimentado suele organizarlos de formas definidas, utilizando un abanico de recursos o modismos; es decir, de alguna forma también está utilizando bloques predefinidos, que podemos considerar componentes. Un bucle que recorre una matriz, un condicional para tratamiento de errores, un intercambio de valores de variables... son ejemplos de estos microcomponentes.

Al considerar el código como un compuesto de microcomponentes, se abren las puertas a la aplicación de Itacio para la verificación automática de código fuente. Ya que la verificación de código escrito *tal cual* parece tremendamente difícil y debe afrontar incluso problemas de computabilidad –como se ha mencionado al hablar de interpretación abstracta-, lo que se propone aquí es *construir el código de manera que sea verificable*, apoyándose en el concepto de componente. De este modo, una simple llamada a función queda salvaguardada por las expresiones restrictivas que irían ligadas a la misma; incluso si las restricciones se satisfacen de manera transitiva o indirecta, gracias a la combinación de instrucciones anteriores –en puntos del programa alejados-, el sistema de inferencias es capaz de tenerlo en cuenta.

Las pruebas realizadas revelan que el modelo puede aplicarse en este ámbito, aunque el nivel de microcomponentes va muy ligado al hecho mismo de programar, y para percibir plenamente su utilidad parece necesario disponer simultáneamente de un sistema de generación de código que, basándose en la descripción de los componentes, genere el programa final. Esto requiere una infraestructura cuyo desarrollo no hemos podido acometer con suficiente profundidad; aunque es nuestra opinión que la construcción de ese sistema es posible (y hemos realizado pruebas preliminares con éxito), lo cierto es que aún no podemos ofrecer aquí ejemplos concluyentes.

Contratos de reutilización

Se ha aplicado el modelo Itacio a los contratos de reutilización (como se describen en [14]). Los contratos de reutilización son especificaciones que describen la forma en que diversos entes colaboran. Un contrato define qué participantes lo integran, qué operaciones realiza cada uno, de qué otros participantes conoce la existencia, y en qué operaciones ajenas se basa un participante para cumplimentar las suyas propias. En otras palabras, define la dependencia entre participantes y entre operaciones de los participantes. Esto se expresa en términos de un sencillo lenguaje (llamado Contracts) definido a tal fin en trabajos anteriores.

Al expresar formalmente un contrato de reutilización, se tiene bajo control (ya que se hace explícito) quién depende de quién y para qué. Más aún, las posibles modificaciones que se realizan a estos contratos se encuentran categorizadas bajo la forma de ocho posibles operadores. De este modo, la evolución temporal de un contrato (adición de operaciones a un participante, eliminación de participantes, etc.) se expresa en términos de operadores aplicados sobre el mismo.

En nuestro caso, el modelo Itacio ha sido aplicado con el fin de verificar la evolución temporal (mantenimiento) de un sistema cuyo comportamiento se haya expresado en forma de contratos de reutilización. Resulta peculiar la instanciación del modelo en este caso; cabría esperar que, de acuerdo con las ideas habituales acerca de componentes, se considerase componente a cada participante del contrato [11]. Pero en este caso no es así; no se pretendía verificar las características estructurales del contrato, sino su evolución *temporal*. Por tanto, lo que se ensambla en este caso son contratos y modificadores, para construir la secuencia que modela dicha evolución temporal; y en este ejemplo se ha considerado que los componentes son, precisamente, los contratos y los operadores que se aplican sobre ellos.

Como de costumbre, una vez establecida la definición de componente / fuente / sumidero sólo restaba escribir las expresiones restrictivas que describen cada componente, lo que constituye una especie de traducción de la notación original usada en Contracts a lógica de primer orden. Tras hacer esto y modelar sistemas concretos haciendo uso del prototipo, se pudo comprobar que el modelo era aplicable a esta situación; por ejemplo, hay un caso de aplicación que permite tener bajo control una modificación peligrosa que –en un programa real- se ha realizado a la biblioteca de clases MFC de Microsoft. Si en una versión posterior de MFC cambiasen las dependencias entre los elementos que la integran de manera que la modificación realizada ya no funcionase, la mera compilación no ofrecería información alguna sobre este hecho, pero el sistema de verificación detectaría que ha dejado de cumplirse una suposición básica del diseño, y señalaría exactamente en qué punto de la evolución temporal tiene influencia el cambio en el código base.

Diagnóstico de equipos y gestión de configuraciones a distancia

Actualmente, el grupo de investigación está trabajando en la aplicación del modelo Itacio al diagnóstico de equipos a distancia. Un problema tradicional en el sistema operativo Windows es la existencia de componentes (librerías) mal instalados, incoherencias o fallos de configuración, incompatibilidad de versiones entre módulos ejecutables, etc. Estos problemas de gestión de configuraciones pueden ser de gran complejidad.

En este caso, parece claro que se puede aplicar también el modelo Itacio. Se trata simplemente de modelar –haciendo uso de las abstracciones habituales en Itacio- el programa que se desea verificar, siendo los componentes los módulos ejecutables, el propio sistema operativo, etc. Puesto que en este caso los “hechos” que se manejan se pueden comprobar mediante código ejecutable que examine físicamente el sistema, basta con implementar librerías adecuadas en un lenguaje imperativo –en este caso C- y dejar que el sistema de inferencia recurra a estas librerías cuando lo necesite, cosa técnicamente posible.

Evidentemente, la instalación de todo el sistema de inferencia en el equipo a diagnosticar podría causar interferencias en la configuración, enmascarando el problema o incluso eliminándolo como efecto lateral. Para evitarlo, se ha desarrollado un servidor o *stub*, un pequeño programa con requisitos de instalación y configuración mínimos –y por tanto con muy baja influencia en el equipo sometido a examen- cuya única misión es atender peticiones remotas –vía TCP/IP- de información sobre la configuración del sistema que se verifica. De este modo, el sistema prototipo de Itacio puede encontrarse instalado en una máquina cualquiera, y desde esa posición actuar evaluando cualquier otra máquina conectada a esta a través del protocolo TCP/IP.

Como ejemplo, se ha preparado un producto –que llamaremos WDMonitor- que pretende monitorizar, a efectos estadísticos, la frecuencia con que los usuarios del procesador de textos Microsoft Word efectúan la acción de “Guardar” durante una sesión de edición. Este sistema debe integrarse con Word, de manera que cierta librería quede instalada en su directorio de inicio; asimismo, cierta plantilla de documento que contiene recursos de programación debe estar también disponible en el directorio de plantillas de usuario de Word.

En la práctica, la instalación de este sistema puede resultar más compleja de lo que parece. Las diferentes versiones de Word almacenan estos elementos en diferentes directorios; además, pueden presentarse problemas de compatibilidad

entre los componentes del sistema de monitorización y el procesador de textos, ya que deben colaborar entre sí, por lo que parece adecuado exigir que el sistema se utilice sólo con las versiones adecuadas de Word. Más aún, los usuarios pueden alterar los directorios de plantillas y de inicio, e incluso ubicarlos en unidades de red, que pueden no estar disponibles en el momento de la instalación o estar protegidas contra escritura. Además, algunos sistemas de instalación, ante la imposibilidad de grabar el fichero en el lugar deseado, pueden resolver que el fichero se instale en un directorio "inocuo", dando al usuario información sobre lo que ha ocurrido y las instrucciones para solucionar el problema manualmente, cosa que por supuesto el usurario puede no hacer; un intento posterior podría conseguir su objetivo, y el equipo tendría componentes repetidos. Como resultado de todo esto, si WDMonitor no funciona puede haber múltiples y variadas causas.

Para solucionar este tipo de problemas, parece adecuado modelar WDMonitor de forma que cada elemento implicado –la librería, la plantilla de documento, el propio Word y el sistema operativo- sean diferentes componentes del sistema. Una vez representadas sus dependencias como conexiones, y redactadas las expresiones restrictivas correspondientes que describen el buen funcionamiento del programa, la verificación del sistema en Itacio provocará que el sistema de inferencias "razone" y examine los diferentes factores implicados, y esto conllevará –de forma automática- la invocación del *stub*, que irá informando al motor de inferencias sobre las distintas características del equipo (versión del sistema operativo, versión de Word, directorio de inicio de Word, si la librería se encuentra allí o no, dónde se encuentra en caso contrario, etc.) Este sistema es capaz de señalar cuál de las interacciones entre los componentes del sistema es incorrecta y por qué.

6 CONCLUSIONES Y LÍNEAS DE INVESTIGACIÓN

A lo largo de este artículo se ha mostrado que un modelo de componentes puede ser utilizado como marco en el que inscribir el conocimiento que se tiene sobre los componentes de un sistema, de modo que ese conocimiento se utilice de manera efectiva con el fin de detectar automáticamente los defectos que surgen de la combinación de dichos componentes. Este sistema, además, presenta las ventajas de una alta simplicidad, flexibilidad y sencillez técnica, de modo que su aplicación en el proceso de desarrollo parece más probable que la de métodos formales que requieren una formación más específica y rigurosa.

Los principales problemas que presenta este sistema son que en algunos casos se requiere la creación de ciertas herramientas adicionales para percibir plenamente sus ventajas y que para un buen funcionamiento del mismo se deben arbitrar medidas para que las expresiones restrictivas de los componentes sean coherentes (en cuanto a nomenclatura y estructura).

En un futuro próximo, el grupo de investigación pretende seguir trabajando en la aplicación de este modelo al diagnóstico remoto de equipos (un área que parece posible llevar incluso a una funcionalidad plena, pese a contar sólo con un prototipo del sistema y no con una herramienta robusta), así como a otros niveles de abstracción en el proceso de desarrollo de software.

BIBLIOGRAFÍA

1. Abd-Allah, A. *Composing Heterogeneous Software Architectures*. Doctoral Dissertation, Center for Software Engineering, University of Southern California, August 1996.
2. Cernuda, A., Labra, J. E., Cueva, J. M. *Itacio: A Component Model for Verifying Software at Construction Time.* III ICSE Workshop on CBSE. 5-6 de Junio de 2000, Limerick, Irlanda. http://www.sei.cmu.edu/cbs/cbse2000/papers/index.html
3. Cernuda, A., Labra, J. E., Cueva, J. M. *A Model for Integrating Knowledge into Component-Based Software Development.* Actas KM - SOCO 2001, 26-29 de Junio de 2001, Paisley (Escocia). ICSC Academic Press, ISBN 3-906454-27-4.
4. Cousot, P., Cousot, R. *Static Determination of Dynamic Properties of Programs*. Proceedings of the 2nd International Symposium on Programming. Editor: B. Robinet. Paris, 13-15 april, 1976.
5. Cousot, P., Halbwachs, N. *Automatic Discovery of Linear Restraints Among Variables of a Program*. Conference Record of the Fifth Annual ACM Symposium on Principles of Programming Languages. Tucson, Arizona, 23-25 january, 1978.
6. Cousot, P. *Abstract Interpretation*. ACM Computing Surveys, Vol. 28, Nº 2, Junio de 1996.

7. ECLiPSe Web site: http://www.icparc.ic.ac.uk/eclipse
8. Frühwirt, T. et al. *Constraint Logic Programming – An Informal Introduction*. Technical report ECRC-93-5. European Computer-Industry Research Centre, February 1993.
9. Gacek, C., Boehm, B. *Composing Components: How Does One Detect Potential Architectural Mismatches?* Position paper to the OMG-DARPA-MCC Workshop on Compositional Software. Center for Software Engineering (University of Southern California) 1998. http://sunset.usc.edu/TechRpts/Papers/usccse98-505.html
10. Garlan, D., Allen, R., Ockerbloom, J. *Achitectural Mismatch or Why it's hard to build systems out of existing parts*. IEEE Software, Noviembre de 1995, págs. 17-26.
11. Hall, Pat. *Educational Case Study – what is the model for an ideal component? Must it be an object?* III ICSE Workshop on CBSE. 22nd ICSE. 5-6 de Junio de 2000, Limerick (Irlanda).
12. Itacio: página web del proyecto. http://i.am/itacio
13. Jones, N.D., Nielsen, F. *Abstract Interpretation: a Semantics-Based Tool for Program Analysis*. 30 june 1994.
14. Lucas, C. *Documenting Reuse and Evolution with Reuse Contracts*. Ph.D. Dissertation, Vrije Universiteit Brussel, Belgium. September 1997.
15. Steyaert, Patrick et al. *Reuse contracts: Managing the evolution of reusable assets.* Proceedings of OOPSLA'96, vol. 31(10) of ACM Sigplan Notices, pages 268-285. ACM Press, 1996.
16. Szyperski, Clemens. *Component Software – Beyond Object-Oriented Programming*. Addison-Wesley, 1997.
17. Wallace, Mark et al. *ECLiPSe: A Platform for Constraint Logic Programming*. William Peney Laboratory, Imperial College, London. August 1997.