Construcción de Representaciones Innovadoras del Dominio del Programa para Facilitar la Comprensión de Programas

Maria Matkovic, Mario Berón, Lorena Baigorria
Facultad de Ciencias Físico Matemáticas y Naturales - Universidad Nacional de San Luis
Ejército de los Andes 950 - San Luis- Argentina
email: mariitamatkovic@gmail.com, {mberon,flbaigor}@unsl.edu.ar

Pedro Rangel Henriques
Departamento de Informática - Universidade do Minho
Braga-Portugal
email: pedrorangelhenriques@gmail.com

Maria J. Pereira
Departamento de Informática - Instituto Politécnico de Bragança
Bragança - Portugal
email: mjoao@ipb.pt

Resumen

La comprensión de programas es una disciplina de la Ingeniería de Software cuyo objetivo es proveer Modelos, Métodos, Técnicas y Herramientas con el propósito de facilitar el estudio y entendimiento de los sistemas de software.

Uno de los desafíos más importantes en el contexto de Comprensión de Programas consiste en Relacionar el Dominio del Problema con el Dominio del Programa. El primero hace referencia a la salida del sistema. El segundo a las componentes de software utilizadas para producir dicha salida. Una manera de construir esta relación consiste en definir una representación para cada dominio y luego establecer un procedimiento de vinculación entre ambas representaciones.

En este artículo se presenta una línea de investigación que estudia diferentes formas de representar el Dominio del Programa. Esta línea de investigación intenta hacer énfasis en aquellas representaciones que faciliten la vinculación con representaciones del Dominio de Problema.

Palabras clave: Comprensión de Programas, Dominio del Problema, Dominio del Programa, Ontología.

1. Contexto

La línea de investigación descripta en este Artículo se encuentra enmarcada en el contexto del proyecto: *Ingeniería del Software: Conceptos Métodos Técnicas y Herramientas en un Contexto de Ingeniería de Software en Evolución* de la Universidad Nacional de San Luis. Dicho proyecto, es reconocido por el programa de incentivos y es la continuación de diferentes proyectos de investigación de gran éxito a nivel nacional e internacional. También forma parte del proyecto bilateral entre la Universidade do Minho (Portugal) y la Universidad Nacional de San Luis (Argentina) denominado *Quixote: Desarrollo de Modelos del Dominio del Problema para Inter-relacionar las Vistas Comportamental y Operacional de Sistemas de Software*. Quixote fue aprobado por el Ministerio de Ciencia, Tecnología e Innovación Productiva de la Nación (MinCyT) y

la Fundação para a Ciência e Tecnología (FCT) de Portugal. Ambos entes soportan económicamente la realización de diferentes misiones de investigación desde Argentina a Portugal y viceversa.

2. Introducción

La Comprensión de Programas es una disciplina de Ingeniería de Software cuyo objetivo es crear Modelos, Métodos, Técnicas y Herramientas, basada en un proceso de aprendizaje y en un proceso de ingeniería, para ayudar a los ingenieros a obtener un conocimiento más profundo sobre los sistemas de software [16, 15, 13].

La comprensión de programas consiste en la habilidad de entender varias unidades o módulos de código escritos en un lenguaje de programación de alto nivel, que integran una aplicación informática. La comprensión de programas está profundamente ligada a la Ingeniería del Software y se necesita para la reutilización, inspección, manutención, reingeniería, migración y extensión de los sistemas de software existentes [5, 12, 7].

Comprender un sistema de software implica construir un mapeo entre las operaciones internas que realiza el programa y los efectos producidos por esas operaciones en el Dominio del Problema.

La creación de este tipo de mapeo requiere de la: i) Definición de una representación del Dominio del Problema; ii) La definición de una representación del Dominio del Programa y iii) La elaboración de un procedimiento de vinculación entre ambas representaciones.

La realización de los pasos previamente mencionados requiere de la recuperación de información desde el sistema y de otras fuentes tales como documentación, entrevistas, etc. Actualmente existen diferentes estrategias de recuperación de información dependiendo de la clase de información que se desee extraer. Si se pretende recuperar información relacionada con el código fuente del sistema entonces las técnicas de compilación tradicionales son las más apropiadas para realizar dicha actividad [5, 1].

Si se busca conocer el comportamiento del sistema en tiempo de ejecución las estrategias de análisis dinámico son las más adecuadas [14, 2, 4, 11].

Finalmente, si se desea extraer información desde otras fuentes textuales se pueden utilizar *Técnicas de Procesamiento de Lenguaje Natural*.

La línea de investigación presentada en este artículo se centra en el estudio y la creación de representaciones innovadoras del Dominio del Programa. Esta tarea tiene como principal objetivo proveer representaciones útiles que faciliten la elaboración de procedimientos que permitan vincular el Dominio del Problema con el Dominio del Programa.

3. Línea de investigación y desarrollo

La línea de investigación descripta en esta sección se basa en el estudio y creación de representaciones del Dominio del Programa. Actualmente existen muchas formas de representar el Dominio del Problema dependiendo del tipo de información que se disponga. De esta manera, si la información que se utiliza es estática, es decir la misma es recuperada desde el código fuente, muchas representaciones del programa pueden ser creadas. Ejemplos de este tipo de representaciones son: *Grafo de Funciones*, *Grafo de Comunicación de Módulos*, *Grafo de Tipos*, *Grafo de Dependencias del Sistema*, etc [8, 9]. Por otra parte, si la información disponible se corresponde con el comportamiento del sistema en ejecución otra clase de representaciones pueden ser elaboradas. Algunos ejemplos de ellas son: *Árbol de Ejecución de Funciones*, *Lista de Funciones Usadas en Tiempo de Ejecución*, *Grafo Dinámico de Llamadas a Funciones*, etc [14, 3]. También se han podido observar en la literatura la existencia de representaciones del Dominio del Programa que usan información estática y dinámica. Este tipo de representaciones son referenciadas en la jerga de Ingeniería de Software con el nombre de *Representaciones del Dominio del Programa Mixtas*. El diseño de la representaciones del Dominio del Programa implica el estudio y creación de estrategias de extracción de la información. De hecho, no es posible construir una representación del programa si no se dispone de un mecanismo para recuperar la información necesaria. Por esta razón, el estudio de este tipo de técnicas también forma parte de esta línea de investigación. En este contexto, la información estática se extrae a través de la utilización de técnicas de compilación tradicionales. Por técnicas de compilación tradicionales se entiende, la utilización de definiciones o traducciones dirigidas por la sintaxis para la extracción de

la información requerida. Si la información que se desea extraer es dinámica, entonces se deben utilizar técnicas de instrumentación de código sea a nivel del lenguaje de programación o bien a nivel lenguaje de máquina. La instrumentación de código consiste en la inserción de sentencias en lugares estratégicos del sistema de estudio con el propósito de extraer información acerca del comportamiento del sistema en tiempo de ejecución. Es importante remarcar que las sentencias insertadas no deben cambiar la semántica del programa [6, 10].

Por último y para finalizar esta breve descripción de la línea de investigación, es importante mencionar que es muy útil combinar ambas clases de información para construir representaciones más robustas del Dominio del Programa.

4. Resultados

En esta sección se describen los resultados alcanzados a través de esta investigación. Los mismos están relacionados con la construcción de representaciones del programa usando ontologias. Este tipo de representaciones son muy interesantes e innovadoras porque permiten vincular los elementos del programa a los del lenguaje de programación. Esta característica, posibilita obtener una descripción más detallada y con más semántica de las componentes del programa porque en este tipo de ontología se vinculan los conceptos del lenguaje de programación y del programa en sí. Esta característica, permite que el usuario pueda navegar entre los conceptos y obtener explicaciones más específicas acerca de las componentes del programa.

La tarea de representar programas empleando ontologías se subdividió en tareas parciales que abordan temáticas tales como: la selección de una conceptualización de ontología, el análisis de las herramientas de especificación de ontologías y el desarrollo de una ontología para el lenguaje Java.

4.1. Concepción de Ontología

Se puede decir que no existe una sola definición de Ontología; sin embargo, la más aceptada es la establecida por Gruber, y difundida por Stuer y colegas:

"... una especificación explicita y formal de una conceptualización compartida"

Especificación explicita implica que una ontología se refiere a un dominio particular, por medio de la explicitación de conceptos, propiedades, valores, relaciones, funciones, etc.

Formal cualquier representación debe expresarse en una ontología mediante una expresión formal, siempre igual, que pueda ser leída o reutilizada por cualquier persona o máquina, con independencia del lugar, de la plataforma o idioma del sistema que lo use.

Por *Conceptualización* se interpreta que la ontología desarrolla un modelo,construido por conceptos, abstracto del dominio que representa atributos, valores y relaciones.

La palabra *Compartida* significa que la ontología ha sido aceptada por todos sus usuarios. No obstante, es prácticamente imposible, que todas las personas pertenecientes a un campo específico, acuerden totalmente. Por tal motivo, se desarrollan ontologías que se refieren a dominios específicos y acotados.

4.2. Editores de Ontologías

Existen numerosos editores para la creación de ontologías. Esto se debe a que los lenguajes para especificar Ontologías son muy complicados y propenso cometer errores cuando se trabaja directamente con ellos.

Los editores analizados hasta el momento son los siguientes:

LinkFactory: Es un editor que se puede utilizar tanto para construir ontologías sencillas, como ontologías muy grandes y complejas. El sistema LinkFactory posee dos componentes muy importantes: i) El LinkFactory Server (actúa de servidor) y ii) El LinkFactory Workbench (está en el lado del cliente). Los dos componentes están desarrollados en Java.

OILEd: Es un editor gráfico de Ontologías. Una característica importante de Oiled es que proporciona un razonador conocido como "FaCT" que se utiliza principalmente para comprobar si una ontología es consistente o no.

OntoEdit: Es un editor gráfico de ontologías. Al igual que los anteriores permite representar o definir una ontología gráficamente. La gran ventaja de ésta herramienta es que además de

posibilitar editar ontologías permite almacenarlas en una base de datos relacional.

Protegé: Es uno de los editores más utilizados por los investigadores. Su principal ventaja consiste en la posibilidad de añadir módulos y plug-ins para aumentar su funcionalidad.

Luego de la realización de pequeños ejemplos con las herramientas mencionadas previamente, se optó por especificar las ontologias con Protégé. Esto se debe a que es una herramienta que posibilita la descripción de clases, propiedades e instancias de una forma simple y práctica. Además provee otras facilidades como por ejemplo: tiene un enriquecido conjunto de operadores *and* (intersección), *or* (unión) y *not*(negación). Se basa en un modelo lógico. Este modelo lógico permite definir conceptos complejos relacionando conceptos más simples mediante estos operadores. Esta semántica formal permite el uso de un razonador para derivar sus consecuencias lógicas, es decir hechos que no están literalmente presentes en la ontología, pero están implicados por la semántica. Entre los principales servicios ofrecidos por un razonador se encuentran: i) Verificar si una clase es subclase de otra clase, y ii) Chequear la consistencia de la ontología.

4.3. Definición de una Ontología del Dominio del Lenguaje de Programación Java

La aproximación utilizada para representar el Dominio del Programa mediante ontolgías consta de dos pasos. El primero consiste en la definición la *Ontología del Lenguaje de Programación*, esta ontologia expresa los conceptos del lenguaje, las relaciones existentes entre ellos y un conjunto de axiomas que tienen como objetivo eliminar ambigüedades e inconsistencias. El segundo paso se relaciona con la aplicación de un proceso que permite decorar la *Ontología del Lenguaje de Programación* con elementos del programa que se está analizando. Por ejemplo, el concepto de *Clase* pertenece a la Ontología del Lenguaje de Programación, si el programa bajo estudio contiene una clase *C* esta puede ser añadida a la ontologia de lenguaje como una instancia de *Clase*. Así, la ontología que surge de incorporar instancias a la ontologia del lenguaje se denomina *Ontología del Programa*. Es importante mencionar que esta ontología, a diferencia de la Ontología del Lenguaje de Programación, se puede crear de forma automática utilizando téncnicas de compilación.

El lenguaje que se utilizó para construir las ontologias mencionadas en el párrafo precedente es el lenguaje Java. Dicho lenguaje fue seleccionado porque es ampliamente usado tanto para el desarrollo de aplicaciones científicas y comerciales, las cuales están, en muchas ocasiones, sujetas a tareas de mantenimiento y evolución.

Actualmente, el desarrollo de la ontología del lenguaje de programación está en un estado inicial. Entre los conceptos del lenguaje Java que se ha especificado se encuentran los siguientes: *Clase*, *Clase Común*, *Clase Abstracta*, *Clase Final*, *Método*, *Constructor*, *Método de Clase*, *Método Final*.

A modo de ejemplo, se muestra la especificación de un concepto, los restantes siguen la misma aproximación.

Concepto: Clase

Definición: Una clase es un molde para la construcción de objetos. Cualquier objeto que se construya a partir de la clase tiene el mismo comportamiento y la misma estructura.

Relaciones:

- Una Clase tiene cero o más Métodos.

Axiomas:

- Una Clase se puede extender a sólo una Clase.
- Una Clase no puede tener modificador *static*.
- Una Clase tiene al menos un constructor.
- La clase Clase es disjunta de la clase Método.

Comentario: Que una Clase sea disjunta de Método significa que ninguna instancia de Método puede ser una instancia de Clase y viceversa. Los axiomas de este tipo son requeridos por las herramientas de especificación de ontologias para el funcionamiento apropiado de los razonadores. Un razonador es un módulo de software que utiliza un conjunto de reglas de inferencia que permiten detectar clases implícitas. Además, este módulo realiza chequeos de consistencia.

Ejemplos: Persona, Animal, Alumno, Docente.

5. Formación de Recursos Humanos

Las tareas realizadas en presente línea de investigación están siendo realizadas como parte del desarrollo de tesis para optar por el grado de Licenciado en Ciencias de la Computación. Se espera a corto plazo poder definir, a partir de los resultados obtenidos en las tesis de licenciatura en curso, tesis de maestría o bien de doctorado, como así también trabajos de Especialización en Ingeniería de Software. Es importante mencionar que tanto el equipo argentino como el portugués se encuentran dedicados a la captura de alumnos de grado y posgrado para la realización de estudios de investigación relacionados con las temáticas presentadas en este artículo. Dichos estudios pretenden fortalecer la relación entre la Universidad Nacional de San Luis y la Universidade do Minho.

Referencias

[1] A. V. Aho, R. Sethi, and J. D. Ullman. *Compilers Principles, Techniques and Tools*. 1986.

[2] C. Bennet, D. Myers, M. A Storey, and D. German. Working with Monster Traces: Building a Scalable, Usable Sequence Viewer. *Program Comprehension through Dynamic Analysis*, 1:1–5, 2007.

[3] M. Berón, P. Henriques, M. Pereira, and R. Uzal. Static and Dynamic Strategies to Understand C Programs by Code Annotation. *European Joint Conference on Theory and Practice of Software - Satellite Event - Open Cert.*

[4] B. Cornelissen and Leon Moonen. Visualizing Similarities in Execution Traces. *Program Comprehension through Dynamic Analysis*, 1:6–10, 2007.

[5] T. Eisenbarth, R. Koschke, and D. Simon. Aiding program comprehension by static and dynamic feature analysis. In *ICSM'01: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, page 602, Washington, DC, USA, 2001.

[6] D.W. Embley. Toward semantic understanding: an approach based on information extraction ontologies. *Proceedings of the fifteenth conference on Australasian database-Volume 27*, pages 3–12, 2004.

[7] S. Fleming, E. Kraemer, R. Stirewalt, L. Dillon, and S. Xie. Refining Existing Theories of Program Comprehension During Maintenance for Concurrent Software. *International Conference on Program Comprehension (ICPC08)*.

[8] J. Gross and J. Yellen. *Graph Theory and Its Applications*. CRC Pres, 1999.

[9] Quante J. Does Dynamic Object Process Graph Support Program Understanding? - A Controlled Experiment. *International Conference on Program Comprehension (ICPC08)*, pages 73–82, 2008.

[10] A. Kuhn, O. Greevy, and T. Girba. Applying Semantic Analysis to Feature Execution Traces. *Program Comprehension through Dynamic Analysis*, 1:48–53, 2005.

[11] Lienhard, A. and Girba, T. and Greevy, O. and Nierstransz, O. Exposing side effects in execution traces. *Program Comprehension through Dynamic Analysis*, 1:10–17, 2007.

[12] M. Petrenko, V. Rajlich, and Vanciu R. Partial Domain Comprehension in Software Evolution and Maintenance. *International Conference on Program Comprehension (ICPC08)*.

[13] V. Rajlich and N. Wilde. The role of concepts in program comprehension. *Program Comprehension, 2002. Proceedings.*, pages 271–278, 2002.

[14] A. Rohatgi, Hamou.Lhadjm A., and J. Rilling. An Approach for Mapping Features to Code Based on Static and Dynamic Analysis. *International Conference on Program Comprehension (ICPC08)*.

[15] A. Von Mayrhauser and A. M. Vans. Program comprehension during software maintenance and evolution. *Computer*, 28(8):44–55, 1995.

[16] A. Walenstein. Theory-based analysis of cognitive support in software comprehension tools. In *IWPC '02: Proceedings of the 10th International Workshop on Program Comprehension*, page 75, Washington, DC, USA, 2002. IEEE Computer Society.