Diseño y Construcción de Lenguajes Específicos del Dominio

Mariano Luzza[(1)], Mario Berón[(1)], Germán Montejano[(1)], Mario Peralta[(1)], Pedro Rangel Henriques[(2)]
[(1)]Departamento de Informática/Facultad de Ciencias Físico Matemáticas y Naturales/Universidad Nacional de San Luis
Ejército de los Andes 950 – San Luis – Argentina
email: {mberon,mluzza,gmonte,mperalta}@unsl.edu.ar
[(2)]Departamento de Informática/Universidade do Minho
Braga-Portugal
email: pedrorangelhenriques@gmail.com

RESUMEN

En la actualidad existen numerosos problemas, que si bien pueden ser solucionados con herramientas de propósito general, es más apropiado abordarlos con aplicaciones específicas para ese dominio. En este contexto se encuentran los Lenguajes Específicos del Dominio.
Un Lenguaje Específico del Dominio es un conjunto reducido de construcciones y operaciones que brindan una mayor expresividad y optimización para un dominio particular.
La construcción de Lenguajes Específicos del Dominio (DSL) no es una tarea trivial ya que implica tres fases muy importantes como lo son: Análisis, Diseño e Implementación.
Si bien estas fases coinciden con un proceso de desarrollo de software tradicional tienen otras particularidades relacionadas con los DSL, es que introducen nuevos desafíos para la investigación y desarrollo de los mismos.
En este artículo se presenta una línea de investigación que tiene por objetivo estudiar las actividades involucradas en el proceso de construcción de DSL.

Palabras Claves: Lenguaje Específico del Dominio, Compiladores, Generador de Aplicaciones.

CONTEXTO

La línea de investigación descripta en este artículo se encuentra enmarcada en dos proyectos de investigación. El primero es *Ingeniería de Software: Conceptos Métodos Técnicas y Herramientas en un Contexto de Ingeniería de Software en Evolución* de la Universidad Nacional de San Luis.
El segundo es un proyecto bilateral entre la Universidad Nacional de San Luis y la Universidade do Minho-Portugal. Dicho proyecto se denomina: *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.

1. INTRODUCCIÓN

La ingeniería de software se encarga de resolver problemas a través de métodos, técnicas y herramientas. Estos componentes están especificados/programados en diferentes tipos de lenguajes de propósito general (GPL) ampliamente usados por su versatilidad para resolver una gran diversidad de problemas. Si bien permiten abordar desarrollos de diferentes aplicaciones, en numerosas ocasiones, su uso no será el más adecuado. Esta afirmación se basa en que las soluciones propuestas producen un código poco entendible, ya que seguramente no está expresado en términos del dominio de la solución, y se necesitan realizar muchas tareas que obscurecen la solución del problema, como llamadas a procedimientos o cálculos que no están en forma primitiva en el lenguaje. Los GPL tienen a su favor un mayor soporte con manuales y herramientas. Los GPL suelen ser de más bajo nivel que los DSL, con sintaxis más complicadas y por ello sujetos a error. Pero puede ocurrir que los usuarios finales del lenguaje, podrían no ser programadores, haciendo que estos tengan que aprender conceptos que no son relevantes al dominio, pero si necesarios para manipular el lenguaje.

Observando el código realizado por los programadores, a veces parece que ocultan información valiosa en una cantidad inmensa de código. Cuando se usa un lenguaje de propósito general, el problema está en el mantenimiento y la evolución del sistema, ya que se debe modificar el código en sí mismo, lo cual puede introducir errores y es además una tarea tediosa y lenta que consume muchos recursos.

Existen algunos dominios que pueden resultar muy difíciles de abordar con lenguajes de propósito general. Un ejemplo de ello son las gramáticas, puede resultar muy sencillo realizar una especificación con BNF y claramente no es natural abordar dicho problema con un GPL. Una solución adecuada a la problemática descripta previamente son los lenguajes específicos del dominio. Estos lenguajes son de más alto nivel, más comprensibles, más fáciles de aprender, con menos sintaxis y por ende menos errores, y hacen más sencilla la correspondencia entre el problema y la solución. Sin embargo su creación y desarrollo no es una tarea trivial y merece ser abordada como una línea de investigación.

Este artículo está organizado como sigue. La sección 2 tiene tres finalidades importantes. La primera explica qué es un DSL, la segunda menciona los usos comunes de los DSL y la tercera muestra ejemplos de DSL. La sección 3 describe los resultados de la investigación. La sección 4 describe la formación de recursos humanos llevada a cabo en el contexto de la línea de investigación presentada en este artículo.

2. LENGUAJES ESPECÍFICOS DEL DOMINIO

2.1. Definición

Un Lenguaje Específico del Dominio (DSL) es un conjunto reducido de construcciones y operaciones que brindan una mayor expresividad y optimización para un dominio particular. Según [HUDAK96] un DSL es “la última abstracción”, que captura precisamente la semántica de un dominio de aplicación. Algunos DSL bien conocidos incluyen SQL y expresiones regulares. Claramente cada uno es mejor que un lenguaje de propósito general para representar operaciones sobre, base de datos y cadenas respectivamente, pero no lo sucede lo mismo cuando se desea describir soluciones fuera de su dominio. Algunas industrias poseen también sus propios DSL. Por ejemplo, en telecomunicaciones, los lenguajes de descripción de llamadas son ampliamente utilizados para especificar la secuencia de estados en una llamada telefónica, y en la industria de viajes, otro lenguaje podría ser usado para describir reservas de vuelo. Otras áreas donde también se podrían usar DSL incluyen el plan de rutas de navegación de un sitio web, diagramas de conexión de componentes electrónicos o un árbol genealógico, entre otras.

2.2. Utilización

Los usuarios de los DSL crean modelos que luego se compilan o se traducen a otros artefactos. Es común hallar DSL cuyo uso es generar código de programa en otro lenguaje o un ejecutable, pero también, los DSL son utilizados para generar otros artefactos como un esquema de visualización para ciertos datos. Cuando se define un DSL, se pueden especificar plantillas que lean un modelo del DSL y generen ejecutables, fuentes de otros lenguajes, archivos de texto u otros artefactos. Por ejemplo, se podría tener una plantilla que tome varías relaciones de parentesco simples como padre, hijo y hermano y genere otras relaciones deducibles como abuelo, sobrino y primo. Generalmente, los DSL son creados cuando un grupo de usuarios (no necesariamente desarrolladores) tienen que generar código similar para varios productos. Por ejemplo, sería útil para un equipo de desarrollo de sitios web, contar con un DSL para generar la navegabilidad de las diferentes páginas y que cree tanto un esqueleto del sitio como una herramienta que verifique dicha navegabilidad. El principal beneficio de los DSL, es que pueden ser más sencillos de entender por los usuarios, tanto del DSL como quizás también de los que usarán el producto generado por el DSL, ya que se manejan conceptos en términos del dominio. Además el código (u otro artefacto generado) es más confiable, gracias a la automatización y refactorización de algunas tareas. En un DSL, la representación es más sencilla, sólo se describe cómo resolver el problema en

términos del dominio y no hay construcciones especiales de código para poder completar el algoritmo como ocurriría en un GPL. Gracias a esto, se vuelve más sencillo también introducir cambios en el código si hay cambios en la especificación del problema. A su vez, quizás no sea necesario un programador para utilizar el DSL, ya que estará en términos del dominio y que cualquier usuario del entorno debería manejar. La habilitación del experto en el dominio aumenta en gran medida la eficiencia del mantenimiento de una aplicación a medida que hay cambios en las especificaciones.

A continuación se describen brevemente un conjunto de DSLs ampliamente usados: i) BNF: Se usa para describir las sintaxis de los lenguajes de programación; ii) BPMN: Business Process Modeling Notation es una notación gráfica para modelar procesos de negocio, en un formato de flujo de trabajo (workflow). El principal objetivo de BPMN es proveer una notación estándar que sea de fácil lectura y entendible por parte de todos los involucrados e interesados del negocio y iii) VHDL: El lenguaje de descripción de hardware VHDL, es otro DSL cuyo propósito es simplificar la tarea de describir el diseño de dichos circuitos.

En la sección siguiente se describen los resultados alcanzados hasta el momento en la línea de investigación.

3. RESULTADOS

Una de las principales tareas desarrolladas en el contexto de la línea de investigación descripta en este artículo se puede mencionar el desarrollo de un DSL para el Proyecto Hoshimi (PH). PH es una plataforma que se utiliza para introducir a la informática a los adolescentes, con el uso de un juego que se desarrolla en el cuerpo humano. Dicho juego tiene como finalidad curar enfermedades con robots muy pequeños denominados *nanobots.*
Los *nanobots* entienden comandos básicos: (i) Moverse, (ii) Detenerse, (iii) Recolectar, (iii) Transferir, (iv) Defender, (v) Autodestruirse, (vi) Abrir PI y (vii) Cerrar PI.
La tarea del jugador es desarrollar una estrategia que representa la inteligencia artificial de los *nanobots*. El desarrollo de las estrategias se puede llevar a cabo usando dos opciones:

1. Empleando el editor visual integrado en la plataforma. La ventaja de este modo es que el lenguaje es sencillo y está acotado al dominio (de hecho es un DSL). La desventaja es la escasa expresividad y funcionalidades disponibles, como el uso de variables, muy necesarias para que los *nanobots* guarden su estado.
2. Usando algún IDE de .Net. La ventaja de este modo es que están accesibles todas los funcionalidades de la API del juego y todas las herramientas de un GPL (por ejemplo C# o VisualBasic.Net) como definición de tipos, uso de estructuras como listas y diccionarios y lo más importante, uso y definición de variables. Como contrapartida, se puede decir que los alumnos deben aprender a programar con lenguajes orientados a objetos y lidiar con errores que entorpecen el aprendizaje como omitir palabras claves o símbolos de puntuación, problemas con los identificadores (escribirlos mal o no respetar las mayúsculas), respetar estructuras del código, etc.

Es así como surge la idea de PH-Asistido, un editor visual para PH que parte de la idea del editor incorporado en la plataforma del Proyecto Hoshimi (ítem 1), y además provee un potencial de trabajo similar al mencionado en el ítem 2. En PH-Asistido se mantienen todos los comandos básicos de los *nanobots*. No obstante algunos de ellos fueron mejorados y/o extendidos. En primer lugar, se flexibilizaron los comandos relativos al control del flujo, como la iteración, que con las modificaciones permite como condición cualquier expresión lógica (antes sólo permitía un entero que correspondía a la cantidad de iteraciones). Esto también se aplica a la decisión (comando similar a un if-then-else) que antes carecía de gran expresividad en el armado de las condiciones. Además, se agregaron nuevas funcionalidades, las más importantes están relacionadas al trabajo con variables. En PH-Asistido es posible definir variables, cambiarles el valor usando una acción de asignación y utilizarlas como parámetros en los demás comandos. Es importante destacar

que la acción de asignación posee un control visual que asiste en el armado de la operación (ver figura 1).

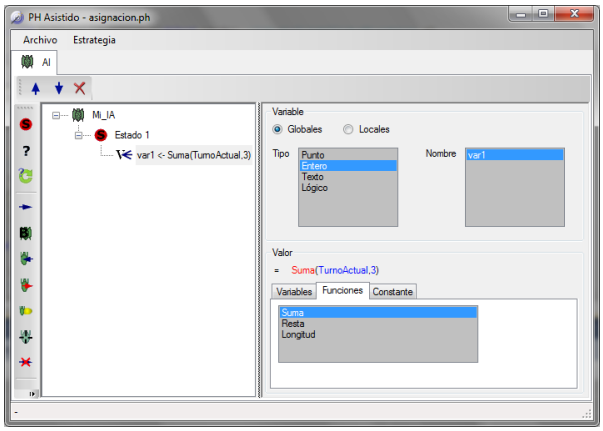

Figura 1 – Parametrización del comando asignación

Por un lado, se especifica el lado izquierdo de la asignación, eligiendo primero el tipo de la variable desde una lista y luego el nombre desde otra lista filtrada por el tipo seleccionado. Esto tiene como consecuencia evitar errores de sintácticos tanto para el tipo como para el nombre. Por otro lado, se especifica el lado derecho de la asignación con una herramienta que permite el armado de expresiones. Esta herramienta tiene tres pestañas: (i) *Variables* que permite seleccionar de un listado de variables filtradas por el tipo correcto que espera la expresión; (ii) *Funciones* que permite seleccionar de un listado de funciones filtrado por aquellas cuyo tipo devuelto sea el esperado por la expresión; y (iii) *Constante* que permite especificar una constante literal. En el caso de que se seleccione una función, el proceso para armar las expresiones para los parámetros, se repite. En otro caso termina. Nuevamente esto tiene como consecuencia evitar errores sintácticos pero también otros tipos errores semánticos como errores de tipos, etc.

Esta construcción de expresiones asistida (CAE) es reutilizada en todos los procesos de parametrización de todos los comandos empleados en la estrategia. De esta forma se garantiza que las expresiones sean válidas al ser construidas.

Si bien las extensiones descriptas previamente son suficientes para mejorar los desarrollos, es posible que el usuario necesite algunas más, que están en la API pero no en el editor. Por ello, se incluyó la posibilidad de extender las funciones y las variables/propiedades que se pueden utilizar en la CAE. Basta con editar el archivo xml correspondiente (Funciones.xml y Variables.xml) y agregar los elementos necesarios. En las figuras 2 y 3 se pueden apreciar los esquemas de los archivos de extensión para las funciones y las variables respectivamente.

```
<!DOCTYPE FUNCIONES [
 <!ELEMENT FUNCIONES (FUNCION*)>
 <!ELEMENT FUNCION (PARAMETROS, CODIGO)>
 <!ATTLIST FUNCION nombre CDATA #REQUIRED>
 <!ATTLIST FUNCION tipo CDATA #REQUIRED>
 <!ELEMENT PARAMETROS (PARAMETRO*)>
 <!ELEMENT PARAMETRO EMPTY>
 <!ATTLIST PARAMETRO nombre CDATA #REQUIRED>
 <!ATTLIST PARAMETRO tipo CDATA #REQUIRED>
 <!ELEMENT CODIGO (#PCDATA)>
]>
```

Figura 2 – DTD del arhivo de extensión de funciónes

```
<!DOCTYPE VARIABLES [
 <!ELEMENT VARIABLES (VARIABLE*)>
 <!ELEMENT VARIABLE (CODIGO)>
 <!ATTLIST VARIABLE nombre CDATA #REQUIRED>
 <!ATTLIST VARIABLE tipo CDATA #REQUIRED>
 <!ATTLIST VARIABLE alcance (global | local)
                    #REQUIRED>
 <!ATTLIST VARIABLE soloLectura (si | no)
                    #REQUIRED>
 <!ELEMENT CODIGO (#PCDATA)>
]>
```

Figura 3 – DTD del arhivo de extensión de varibles/propiedades

```
<FUNCION nombre="Modulo" tipo="Entero">
 <PARAMETROS>
  <PARAMETRO nombre="x" tipo="Entero">
  <PARAMETRO nombre="y" tipo="Entero">
 </PARAMETROS>
 <CODIGO> return x%y; </CODIGO>
</FUNCION>
```

Figura 4 – Ejemplo de extensión de función

```
<VARIABLE nombre="EstaVivo" tipo="Logico"
          alcance="local" soloLectura="si">
 <CODIGO>
  { { get return HitPoints > 0; } }
 </CODIGO>
</FUNCION>
```

Figura 5 – Ejemplo de extensión de variable

Para agregar una función basta con añadir un elemento *FUNCION* y establecer su: *nombre, tipo, parámetros* y *código* C# correspondiente. Para agregar una variable o propiedad, basta con añadir un elemento *VARIABLE* y establecer: *nombre, tipo* y *código C#* correspondiente. Las figuras 4 y 5 muestran sendos ejemplos de extensión de función y variable respectivamente.

Como trabajo futuro a corto plazo se pretende

añadir otras funcionalidades, como por ejemplo la incorporación de tipos estructurados para variables tales como listas y diccionarios y los correspondientes comandos para manipularlas. Además, se pretende reutilizar la estrategia empleada en el desarrollo de este editor para crear un editor cuyo lenguaje sea parecido al utilizado en cursos de introducción a la programación. Dichos cursos se imparten, generalmente en los primeros años de las carreras de informática. Estos lenguajes rara vez poseen un editor y un compilador/intérprete que permita ejecutar el código especificado por los estudiantes. Debido a esto los alumnos siempre trabajan en papel y no pueden ver los resultados de sus desarrollos. Por consiguiente, el proceso de comprensión de programas se torna más difícil.

4. FORMACIÓN DE RECURSOS HUMANOS

Las tareas llevadas a cabo en esta línea de investigación están siendo realizadas en diferentes tesis correspondientes a la Licenciatura en Ciencias de la Computación. Se proyecta a corto plazo la continuación de esta investigación con la definición de nuevas tesis de licenciatura, tesis de maestría y doctorado.

5. REFERENCIAS BIBLIOGRÁFICAS

[1] P. Hudak. Building domain-specific embedded languages. ACM Computing Surveys, 28(4es), December 1996.

[2] M. Mernik, J. Heering, T. Sloane. When and How to Develop Domain-Specific Languages. ACM Computing Surveys, 37(4es), December 2005.

[3] J. Sanchez Cuadrado, J. García Molina. A Model-Based Approach to Families of Embedded Domain-Specific Languages. IEEE Transactions on Software Engineering, vol 35 Nº 6, November/December 2009.

[4] A. van Deursen, P. Klint, J. Visser. Domain-Specific Languages: An Annotated Bibliography. ACM Sigplan Notices, Vol. 35, No. 6, 2000.

[5] A. van Deursen, P. Klint. Domain-Specific Language Design Requires Feature Description. CIT Journal of Computing and Information Technology, Special Issue on Domain-Specific Languages, Part II, Eds. R. Laemmel, M. Mernik, Vol. 10, No. 1, pages 1-17, 2002.

[6] T. Kosar, P. Martinez, P. Barrientos, M. Mernik. A preliminary study on various implementation approaches of domain-specific languages. Inf. Softw. Technol., Vol. 50, No. 5, 2008.

[7] D. Wile. Lessons learned from real DSL experiments. Science of Computer Programming, Vol. 51, No. 3, 2004.

[8] Francisco P. Andrés, Juan de Lara, and Esther Guerra. Domain Specific Languages with Graphical and Textual Views. In Andy Schürr, Manfred Nagl, and Albert Zündorf, editors, AGTIVE, volume 5088 of Lecture Notes in Computer Science, pages 82–97. Springer, 2007.

[9] F. Arefi, C.E. Hughes, and D.A. Workman. The object-oriented design of a visual syntax-directed editor generator. In Computer Software and Applications Conference, 1989. COMPSAC 89., Proceedings of the 13th Annual International, pages 389 –396, sep 1989.

[10] Farah Arefi, Charles E. Hughes, and David A. Workman. Automatically generating visual syntaxdirected editors. Commun. ACM, 33:349–360, March 1990.

[11] Daniela da Cruz, Ruben Fonseca, Maria Joao Varanda Pereira, Mario Beron, and Pedro Rangel Henriques. Comparing generators for language-based tools. In CoRTA-07 - Compiler Related Techonlogies and Applications, Covilhã, Portugal, July 2007.

[12] Stavroula Georgantaki and Symeon Retalis. Using educational tools for teaching object oriented design and programming. Journal of Information Technology Impact (Jiti), 7(2):111–130, 2007.

[13] Pedro Henriques, Maria Joao Varanda, Marjan Mernik, Mitja Lenic, Jeff Gray, and Hui Wu. Automatic generation of language-based tools using lisa system. IEE Software Journal, 152(2):54–70, April 2005.

[14] J. Maloney, M. Resnick, N. Rusk, B. Silverman, and E. Eastmond. The scratch programming language and environment. ACM Transactions on Computing Education, 10(4):1–15, 2010.