Implementación en C# de las reglas heurísticas de conversión de esquemas preconceptuales a diagramas uml 2.0

Heuristic rules for transforming preconceptual schemas into uml 2.0 diagrams: a C# implementation

Carlos Zapata[*]*, David Cardona*

Grupo de Investigación en Ingeniería de Software. Escuela de Sistemas. Facultad de Minas. Universidad Nacional de Colombia, sede Medellín. Calle 59A No 63 - 20 Medellín - Colombia

(Recibido el 20 de junio de 2007. Aceptado el 29 de enero de 2008)

Resumen

Desde mediados de los años noventa se viene gestando una tendencia que busca la generación automática de esquemas conceptuales de UML a partir de lenguajes controlados, empleando para ello reglas heurísticas de transformación. En esta tendencia, aún subsisten algunos problemas: ambigüedad en las reglas, procesos semiasistidos con alta participación de expertos, dificultades en la representación conjunta de estructura y dinamismo del mundo, generación de un solo diagrama y carencia de implementación de algunos trabajos. En este artículo se emplean los denominados Esquemas Preconceptuales como lenguaje gráfico de representación del dominio de una aplicación de software, y se implementan en C# las reglas de conversión de esos esquemas a diagramas de UML 2.0. La implementación en C# elimina la ambigüedad que podrían contener las reglas y permiten la automatización del proceso sin la mediación de expertos. Finalmente, se muestra mediante un caso de estudio el funcionamiento de un prototipo en C# basado en estas reglas.

---------- *Palabras clave:* Esquema preconceptual, diagramas de clases, diagramas de comunicación, diagramas de máquina de estados, UML 2.0, reglas de transformación, lenguaje de programación C#.

* Autor de correspondencia: teléfono: + 57 + 4 + 425 53 74, fax: + 57 + 4 + 425 52 27, correo electrónico: cmzapata@unalmed.edu.co (C. M. Zapata)

Abstract

From the mid-nineties, a new path for automatically generating UML conceptual schemas from controlled languages, by means of heuristic rules, has been proposed. This path still exhibit problems to be solved: ambiguity of heuristic rules, semi-automated expert-participation processes, difficulties in representing structural and dynamic features of the domain, focus on only one diagram, and lack of implementation. In this paper, we employ the so-called Pre-conceptual Schemas as a graphical language for representing the domain of a software application; also, we implement in the C# language the conversion rules from Pre-conceptual Schemas into UML 2.0 diagrams. C# implementation avoids ambiguity of some of the conversion rules; furthermore, C# implementation permits process automation without the need of experts. Finally, we show the functionality of C# rule-based prototype by means of a case study.

---------- *Keywords:* Pre-conceptual schemas, class diagram, communication diagram, state machine diagram, UML 2.0, transformation rules, C# programming language.

Introducción

Los esquemas conceptuales se suelen emplear para el modelado de los diferentes dominios de aplicación, con el fin de elaborar posteriormente una aplicación de software a partir de ellos. Uno de los principales lenguajes para la elaboración de esquemas conceptuales es el denominado Lenguaje Unificado de Modelado (UML por sus siglas en inglés) creado mediados de los años noventa [1]. Si bien desde los años ochenta existen propuestas para generar esquemas conceptuales desde subconjuntos del lenguaje natural [2], el surgimiento de UML ha motivado el surgimiento de muchos trabajos en esta tendencia, buscando obtener los diagramas de UML de manera automática o semiautomática. Para lograr ese fin, en los diferentes trabajos se han propuesto reglas heurísticas para posibilitar la transformación de los discursos en lenguaje natural controlado en esquemas conceptuales [2, 3, 4, 5, 6, 7].

En esta nueva tendencia se han presentado grandes progresos, pero los esfuerzos aún se justifican porque los diferentes trabajos realizados exhiben limitaciones tales como:

- Las dificultades para expresar de manera conjunta la estructura y el dinamismo del mundo para la generación de diferentes tipos de esquemas conceptuales [6, 7].
- La generación de un único esquema conceptual a partir del discurso, como es el caso del diagrama entidad-relación [2, 3] o el diagrama de clases [4].
- La ambigüedad que se presenta en la definición y aplicación de las reglas heurísticas [2].
- La necesidad de asistencia total o parcial en el proceso por parte de expertos, lo cual dificulta la automatización total del mismo [3, 4, 5].
- La carencia de implementación de algunos de estos trabajos [2, 6].

Como una forma de solución a estos problemas, en este artículo se emplea un trabajo previo del grupo de Investigación en Ingeniería de Software de la Universidad Nacional de Colombia, los denominados "Esquemas Preconceptuales" [8] y el conjunto de reglas definidas para su traducción a tres diagramas de UML (clases, comunicación y máquina de estados), y se define una implementación de estos elementos en un lenguaje de programación (en este caso C#). Con los Esquemas Preconceptuales y sus reglas heurísticas de transformación se atacan dos de los problemas mencionados (las dificultades para expresar dinamismo y estructura de manera conjunta y la generación de únicamente un diagrama) y con la implementación en C# se solucionan los restantes.

La estructura de este artículo es la siguiente: en la Sección 2, se revisan los trabajos previos para la obtención de diagramas UML a partir de lenguajes controlados; en la Sección 3, se presentan los Esquemas Preconceptuales y los diagramas de UML que se pueden obtener a partir de ellos; en la Sección 4, se presentan las reglas heurísticas que posibilitan el paso de Esquemas Preconceptuales a diagramas de UML y la implementación en *C#* de esas reglas; en la Sección 5, se presenta un caso de estudio; finalmente, en las Secciones 6 y 7, se enuncian las conclusiones y el trabajo futuro respectivamente.

Obtención de esquemas conceptuales a partir de lenguaje natural controlado empleando reglas heurísticas

Los trabajos previos que procuran la obtención de esquemas conceptuales a partir de lenguaje natural se pueden clasificar en tres grupos: los que obtienen esquemas conceptuales de tipo relacional, en este caso el diagrama entidad-relación, los que obtienen especificaciones orientadas a objetos, diferentes de UML, y los que generan diagramas de UML. A continuación se examinan las ventajas y desventajas de esos trabajos.

Reglas heurísticas para la obtención del diagrama Entidad-Relación

Chen propuso un conjunto de once (11) reglas heurísticas para facilitar la obtención del modelo E-R (entidad-relación) a partir de un discurso en lenguaje natural en inglés [2]. Más que reglas, como el mismo Chen anota, lo que presenta en su trabajo son sugerencias sobre cuáles elementos de un discurso pueden ser considerados entidades, relaciones o atributos. Además, enuncia algunas reglas que permiten la transformación de ciertas frases del discurso, por ejemplo, "hay 200 empleados en el departamento" se transformaría en "departamento tiene 200 empleados", identificando dos posibles entidades (empleado, departamento) y una relación entre éstas. El trabajo de Chen presenta algunas desventajas como:

- Son reglas de tipo estructural, ya que generan un diagrama estructural como es el E-R, en el cual no se puede representar adecuadamente el dinamismo del mundo.
- Algunas reglas son ambiguas entre sí, por ejemplo una regla puede definir un sustantivo como atributo y simultáneamente otra lo puede definir como entidad.
- La aplicación de las reglas requiere conocimiento del dominio.
- Fue sólo una propuesta teórica que no se llegó a implementar.

Un segundo trabajo que procura la obtención del diagrama entidad-relación es el de Gangopadhyay [3], el cual pretende obtener el modelo conceptual de una base de datos a partir de una especificación en lenguaje natural usando la teoría de dependencias conceptuales. Su principal desventaja consiste en que define tan sólo la estructura y no procura obtener la parte dinámica; además, pese a que se realizó la implementación de las reglas, el proceso debe ser asistido por un experto del área, con el fin de determinar adecuadamente las dependencias entre los diferentes conceptos presentes en el discurso.

Obtención de especificaciones orientadas a objetos, previas a UML

Bryant [4] busca la especificación de los requisitos de una aplicación de software basándose en las ideas del lenguaje Two-Level Grammar o TLG [9], usando un proceso iterativo en el cual se va dando una transformación a partir de lenguaje natural; en este trabajo, se argumenta que este método permite la obtención de un diseño completo del sistema e inclusive una implementación en código de éste usando una herramienta llamada SDE (Entorno de Desarrollo de Especificaciones por sus siglas en inglés). Entre las ventajas de esta propuesta se puede mencionar el hecho de que se obtiene un diseño orientado a objetos del sistema y se genera código compilable y ejecutable a partir de ese diseño. Sin embargo, este trabajo posee algunas desventajas:

- Es necesario que el interesado especifique todos los requisitos del software hasta que "cada función se haya especificado completamente" [4], lo que indica que el proceso no puede ser totalmente automatizado, puesto que se requiere la mediación de un experto del dominio para culminar la especificación.
- La implementación que se realizó de la propuesta cubre únicamente aspectos estructurales del diseño orientado a objetos y no se involucra el dinamismo que aportan los diagramas de interacción.

Un segundo trabajo en esta dirección [6] busca cerrar la brecha entre la parte lingüística y los esquemas conceptuales por medio de la lógica de predicados y la teoría de conjuntos. Este enfoque es un método formal, lo que le da ventaja sobre muchos otros trabajos que se basan en reglas heurísticas, sin embargo tiene algunas dificultades:

- Su enfoque sirve sólo cómo guía al analista para indicarle cómo elaborar los modelos conceptuales, porque no llegó a implementarse.

- Los modelos conceptuales obtenidos por el analista no siguen la notación UML. En este trabajo se obtienen el diagrama OMT de Rumbaugh y el diagrama de comportamiento de Martin. Ambos diagramas son precursores del UML.
- Se debe realizar una separación previa de la parte estática y la parte dinámica del discurso, antes de la generación de cada diagrama. Al hacer esto, se puede perder información valiosa para los dos diagramas y, consecuentemente, se pueden generar problemas de completitud y consistencia en los diagramas resultantes.

Obtención de diagramas UML a partir de lenguajes controlados

La propuesta de Konrad y Cheng [5] busca validar un modelo UML a través del análisis del discurso que le dió origen; para ello, los autores proponen una formalización de las fases de especificación y análisis. Este proceso tiene varios aspectos positivos, por ejemplo el hecho de que se usen patrones para la validación de los modelos, que pueden ser reutilizados en futuras validaciones; además, este proceso se implementó en una aplicación de software. Sin embargo, presenta las siguientes desventajas:

- Se requiere un experto en métodos formales tanto para agregar nuevos patrones como para adaptar los ya existentes a un dominio diferente, lo cual hace que el proceso deba ser asistido y no pueda automatizarse en su totalidad.
- No se automatiza la generación del modelo UML, sino únicamente una validación del mismo.

Un segundo ejemplo en esta tendencia lo constituye el proyecto NIBA [7], que busca definir un proceso automático para la obtención de varios diagramas de UML tomando como punto de partida los requisitos del sistema escritos en un lenguaje natural controlado. Para lograr su objetivo, el proyecto NIBA usa un esquema intermedio llamado KCPM (Modelo Conceptual de Prediseño de Klagenfurt por sus siglas en inglés) el cual está compuesto de diferentes artefactos a partir de los cuales se pueden especificar los aspectos dinámicos y estructurales del sistema. La principal desventaja de este enfoque es que el KCPM no es unificado y posee diferentes artefactos según el diagrama que se espera obtener. En otras palabras, hay que elaborar unos artefactos para obtener la parte estructural (diagrama de clases, por ejemplo) y otros para la parte dinámica (diagrama de actividades, por ejemplo), lo cual puede conducir a errores de consistencia y completitud de la información que se desprende del discurso.

Los esquemas preconceptuales y los diagramas de UML 2.0

En la Sección anterior se identificaron varios problemas asociados con la generación automática de esquemas conceptuales de UML a partir de un lenguaje controlado; algunos de esos problemas se superan con los denominados Esquemas Preconceptuales [8], cuyas componentes se pueden visualizar en la Figura 1 y tienen los siguientes significados:

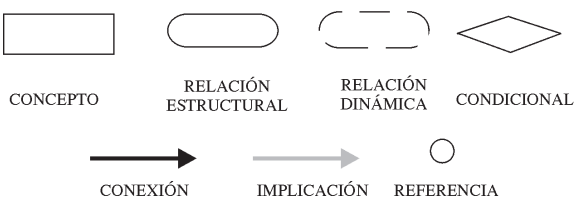

Figura 1 Símbolos básicos de los Esquemas Preconceptuales

- Los conceptos se suelen asociar con sustantivos. Cada concepto sólo puede aparecer una vez en un Esquema Preconceptual.
- Las relaciones estructurales son del tipo “es” y “tiene”.
- Las relaciones dinámicas son verbos que denotan acciones (tales como “registra”, “paga”, “cumple”, “asigna”, “suspende”, “solicita”, etc.).

- Los condicionales son requisitos que se deben cumplir para que se realice una relación dinámica (por ejemplo "usuario.multa>0", "recurso.estado='extraviado'", etc.).
- Las conexiones son vínculos entre conceptos y relaciones (dinámicas o estructurales) o entre relaciones y conceptos.
- Las implicaciones son vínculos del tipo causa-efecto entre relaciones dinámicas o entre condicionales y relaciones dinámicas.
- Las referencias son elementos para unir conceptos que se encuentran físicamente distantes en un esquema preconceptual.

Los esquemas preconceptuales se escriben en forma de triadas, que son conjuntos concepto_fuente-relación (dinámica o estructural)-concepto_destino; por ejemplo "Bibliotecario suspende cuenta" o "Persona tiene nombre".

La forma en que los esquemas preconceptuales resuelven los dos problemas iniciales, que se listaron en la introducción para la generación automática de esquemas conceptuales a partir de discursos en lenguaje natural controlado, es la siguiente:

- Estos esquemas permiten la coexistencia de elementos de tipo estructural y de tipo dinámico en su sintaxis, de forma que el discurso se puede expresar gráficamente de manera cercana a como se haría en lenguaje natural controlado. En la Figura 1, los conceptos y las relaciones estructurales permiten definir la parte estructural, en tanto que las relaciones dinámicas, las implicaciones y los condicionales permiten expresar el dinamismo y el comportamiento de los conceptos. Además, los Esquemas Preconceptuales se pueden obtener de un discurso expresado en una forma de lenguaje natural controlado, que se denomina UN-Lencep [10].
- Los esquemas preconceptuales poseen 14 reglas heurísticas para la conversión a tres diagramas de UML (y no a un solo diagrama como es el caso de varios de los trabajos analizados), que son los diagramas de clases, comunicación y máquina de estados [11].

En síntesis, los esquemas preconceptuales son una representación intermedia entre el lenguaje natural y los esquemas conceptuales de UML. La selección de los diagramas que se pueden obtener a partir de los Esquemas Preconceptuales se debe a las siguientes razones [12]:

- El diagrama de clases de UML se usa para describir los tipos de objetos en un sistema y las relaciones entre estos; además muestra las propiedades y las operaciones de las clases y qué restricciones hay para las relaciones entre objetos, lo que permite modelar la parte estructural de un dominio.
- El diagrama de comunicación es un diagrama de interacción que hace énfasis en las conexiones entre los objetos de un sistema y la secuencia en que estas conexiones se producen, suministrando una idea del dinamismo del dominio.
- El diagrama de máquina de estados es una técnica para describir el comportamiento de un objeto del sistema; está compuesto principalmente por estados y transiciones, donde una transición lleva al objeto de un estado a otro. En este caso se suministra una idea clara del comportamiento de un objeto.

Estas características—estructura, dinamismo y comportamiento—son las recomendadas para analizar en profundidad la especificación de una aplicación de software que se pretenda construir [1].

Implementación en C# de las reglas de transformación de esquemas preconceptuales en diagramas de UML 2.0

Los últimos tres problemas listados en la Introducción de este artículo se ligan con la manera como se implementan las reglas de transformación, con el fin de generar los esquemas conceptuales respectivos. En este sentido, las reglas heurísticas que acompañan a los Esquemas Preconceptuales

tienen una única manera de aplicación, especialmente porque se expresan en términos del lenguaje C#, por lo cual eliminan completamente la ambigüedad. Además, para su aplicación no se requiere ningún tipo de asistencia de parte del analista o de cualquier otro experto. Finalmente, estas reglas se encuentran implementadas en un prototipo que genera, a partir de un discurso expresado en un Esquema Preconceptual, los diagramas de clases, comunicación y máquina de estados de UML 2.0.

Si bien se implementaron en su totalidad las 14 reglas enunciadas por Zapata *et al.* [11], por razones de espacio, en la Tabla 1 sólo se enuncian algunas de ellas, seguidas por el respectivo código en C# que las implementa. La nomenclatura de las reglas corresponde al trabajo de Zapata *et al.* [11].

Tabla 1 Código en C# de las reglas para la generación de esquemas UML a partir de esquemas preconceptuales

Regla	*Código*

4.1.2 El concepto fuente de una relación "tiene" es una clase candidata y el concepto destino será un atributo del concepto fuente.	foreach (ElementoPreconceptual elem in conexionesDiagrama)
	{
	obj = elem as Conexion;
	if (obj != null)
	{
	rel = obj.Inicio as RelacionEstructura;
	con = obj.Fin as Concepto;
	if (!(rel != null && con != null))
	continue;
	if (!(rel.Nombre.ToLower().Equals("es")))
	continue;
	foreach (ElementoPreconceptual elem2 in conexionesDiagrama)
	{
	Conexion con2 = elem2 as Conexion;
	Implicacion impl = elem2 as Implicacion;
	Condicional cond = elem2 as Condicional;
	if (impl != null || cond != null)
	continue;
	herencia = new Herencia(rel.Id);
	if (con2.Fin.Id == rel.Id && con2.Inicio is Concepto)
	{
	claseMadre = new Clase(con.Id);
	claseMadre.Nombre = con.Nombre;
	claseHija = new Clase(con2.Inicio.Id);
	claseHija.Nombre = con2.Inicio as Concepto).Nombre;
	herencia.Inicio = claseHija;
	herencia.Fin = claseMadre;
	this.Herencias.Add(herencia);
	this.agregarClase(claseHija);
	this.agregarClase(claseMadre);
	}}}}

Regla	*Código*

4.1.3 Los conceptos fuente y destino de una relación "es" son clases candidatas entre las cuales se genera una relación de herencia en la cual el concepto destino es la clase padre y el concepto fuente es la clase hija.	foreach (ElementoPreconceptual elem in conexionesDiagrama)
	{
	obj = elem as Conexion;
	if (obj != null)
	{
	rel = obj.Inicio as RelacionEstructura;
	con = obj.Fin as Concepto;
	if (!(rel != null && con != null))
	continue;
	if (!(rel.Nombre.ToLower().Equals("es")))
	continue;
	foreach (ElementoPreconceptual elem2 in conexionesDiagrama)
	{
	Conexion con2 = elem2 as Conexion;
	Implicacion impl = elem2 as Implicacion;
	Condicional cond = elem2 as Condicional;
	if (impl != null || cond != null)
	continue;
	herencia = new Herencia(rel.Id);
	if (con2.Fin.Id == rel.Id && con2.Inicio is Concepto)
	{
	claseMadre = new Clase(con.Id);
	claseMadre.Nombre = con.Nombre;
	claseHija = new Clase(con2.Inicio.Id);
	claseHija.Nombre = con2.Inicio as Concepto).Nombre;
	herencia.Inicio = claseHija;
	herencia.Fin = claseMadre;
	this.Herencias.Add(herencia);
	this.agregarClase(claseHija);
	this.agregarClase(claseMadre);
	}}}}

4.1.5 (Parcial): Una relación dinámica R1 entre un concepto fuente identificado como clase candidata C1 y un concepto destino identificado como atributo A1 de una clase candidata C2 previamente identificada, genera una asociación entre C1 y C2, y una operación en C2 que tendrá el mismo nombre que R1 y recibirá un argumento del mismo tipo que A1	foreach (ElementoPreconceptual elem in conexionesDiagrama)
	{
	obj = elem as Conexion;
	if (obj != null)
	{
	rel = obj.Inicio as RelacionActividad;
	con = obj.Fin as Concepto;
	if (rel == null || con == null)
	continue;
	contenido = rel.Nombre.ToLower();
	aux = this.esAtributo(con.Id);
	if (aux == null)

Regla	*Código*

.	continue;
	operacion = new Operacion(rel.Id);
	operacion.Nombre = contenido.ToLower();
	foreach (ElementoPreconceptual elem2 in conexionesDiagrama)
	{ Conexion con2 = elem2 as Conexion;
	Implicacion impl = elem2 as Implicacion;
	Condicional cond = elem2 as Condicional;
	LineaValores con3 = elem2 as LineaValores;
	if (impl != null || cond != null || con3 != null)
	continue;
	claseFuente = this.esClase(con2.Inicio.Id);
	if (con2.Fin.Id == rel.Id && con2.Inicio is Concepto)
	{ if (aux.Operaciones.ContainsKey(rel.Id))
	continue;
	parametrosOperacion.Add(new AuxiliarRegla(rel.Id,
	aux, con, rel.Nombre.ToLower()));
	if(claseFuente != null)
	{ asoc = new Asociacion(con2.Id);
	asoc.Inicio = claseFuente;
	asoc.Fin = aux;
	this.asociaciones.Add(asoc);
	} } }
	if (parametrosOperacion.Count > 0)
	{
	this.agregarOperacionesConParametros(parametrosOperacion);
	parametrosOperacion.Clear();
	} } }

4.1.7 Si el concepto destino de una relación "tiene" es definido como una clase candidata por otra regla, entonces el concepto fuente de la relación se define como una clase candidata y se genera una relación de agregación en la cual el concepto fuente es el todo y el concepto destino la parte.	foreach (Clase clase in this.ClasesCandidatas)
	{
	foreach (Clase clase2 in this.ClasesCandidatas)
	{
	indice = clase2.indiceDeAtributo(clase.Id);
	if (indice != -1)
	{
	clase2.Atributos.RemoveAt(indice);
	agr = new Agregacion(this.idDesambiguacionActual++);
	agr.Inicio = clase;
	agr.Fin = clase2;
	this.Agregaciones.Add(agr);
	}
	}
	}

Regla	*Código*

4.2.3 (Parcial): Todo concepto en la triada destino de un condicional será una clase candidata.	foreach (ElementoPreconceptual elem in this.elementosDiagrama.Values)
	{
	condicional = elem as Condicional;
	if (condicional != null)
	{
	//condicion = condicional.Nombre;
	foreach (ElementoPreconceptual elemento in this.conexionesDiagrama)
	{
	conexion = elemento as Implicacion;
	if (conexion == null)
	continue;
	else
	{
	if (conexion.Inicio.Id == condicional.Id)
	{
	// conexCondi es una conexión que está
	//conectada
	// a un condicional, puede tener un texto “no”
	conexCondi = conexion;
	relacion = conexion.Fin as RelacionActividad;
	if (relacion != null)
	{
	foreach (ElementoPreconceptual conex in this.conexionesDiagrama)
	{
	conexion2 = conex as Conexion;
	if(conexion2 == null)
	continue;
	else if(conexion2.Inicio.Id == relacion.Id)
	{
	auxiliar = conexion2.Fin as Concepto;
	if (auxiliar != null)
	{
	oper = new Operacion(relacion.Id);
	oper.Nombre = relacion.Nombre.ToLower();
	if ((claseObjetivo = this.esAtributo(auxiliar.Id)) != null)
	{
	if (!claseObjetivo.Operaciones.ContainsKey(oper.Id))
	claseObjetivo.Operaciones.Add(oper.Id, oper);
	listaParametros.Add(new AuxiliarRegla(oper.Id, claseObjetivo, auxiliar, oper.Nombre));
	}
	else
	{
	claseObjetivo = new Clase(auxiliar.Id);
	claseObjetivo.Nombre = auxiliar.Nombre;
	claseObjetivo.Operaciones.Add(oper.Id, oper);

Regla	*Código*

```
                auxiliar = null;
               }
               }
              }
              if (conexion2 != null && conexion2.Fin.Id == relacion.Id)
              {
               auxiliar = conexion2.Inicio as Concepto;
               if (auxiliar != null)
               {
               idAsoc = conexion2.Id;
               claseFuente = new Clase(auxiliar.Id);
               claseFuente.Nombre = auxiliar.Nombre;
               auxiliar = null;
               }
              }
              if (claseFuente != null && claseObjetivo != null)
               break;
             }
             if (claseObjetivo != null)
              this.agregarClase(claseObjetivo);
             if (claseFuente != null)
              this.agregarClase(claseFuente);
             if (claseFuente != null && claseObjetivo != null)
             {
              asoc = new Asociacion(idAsoc);
              asoc.Fin = claseObjetivo;
              asoc.Inicio = claseFuente;
              this.asociaciones.Add(asoc);
             }
             claseFuente = null;
             claseObjetivo = null;
             }
             foreach (AuxiliarRegla temp in listaParametros)
             {
              foreach(Clase claseTemp in this.clasesCandidatas)
             {
              if (claseTemp.Id == temp.clase.Id)
              {
               foreach(Operacion operTemp in claseTemp.Operaciones.Values)
               {
                if(operTemp.Id == temp.idRelAct)
                 operTemp.agregarParametro(temp.concepto.Nombre);
               }
              }
             }
             }
             listaParametros.Clear();
            }
           }
           }       this.desambiguarClases();
          }
         }
```

Regla	*Código*

4.3.1 y 4.3.2 Los mensajes identificados en el diagrama de comunicación, expresados en participio pasado, son candidatos a estados del diagrama de máquina de estados. Además, la secuencia en que se suceden esos estados depende de los números de secuencias identificados en el diagrama de comunicación.	public void identificarElementos9()
	{
	CambioEstado cambioTemp = null;
	Estado estadoTemp = null;
	List<Estado> listaEstados = null;
	List<CambioEstado> listaCambios = null;
	string nombre = "";
	int estadoActual = -1;
	foreach (DiagramaComunicacion diagrama in this.DiagramasDeComunicacion)
	{
	// Armamos el DTE de cada objeto
	foreach (Objeto objeto in diagrama.Objetos)
	{
	listaEstados = new List<Estado>(7);
	listaCambios = new List<CambioEstado>(10);
	nombre = objeto.NombreCompleto.Trim(new char[] { ':' });
	estadoActual = 1;
	foreach (Comunicacion comuni in diagrama.Comunicaciones)
	{
	if (nombre.Equals(comuni.Receptor.NombreCompleto.Trim(new char[] { ':'
	}))
	{
	estadoTemp = new Estado(comuni.Id);
	estadoTemp.NombreEstado = comuni.mensajeSolo.ToUpper() + "DO";
	estadoTemp.numeroEstado = estadoActual++;
	cambioTemp = new CambioEstado(comuni.Id * 100);
	cambioTemp.PreCondiciones = comuni.precondicion;
	cambioTemp.Fin = estadoTemp;
	this.agregarEstado(listaEstados, estadoTemp);
	this.agregarCambioEstado(listaCambios, cambioTemp);
	estadoTemp = null;
	cambioTemp = null;
	}
	}
	diagramas.Add(new DTE(listaEstados, listaCambios, "DTE " + nombre));
	nombre = "";
	estadoTemp = null;
	cambioTemp = null;
	}
	}

Los enunciados de las 14 reglas se pueden consultar en Zapata *et al.* [11]. Todas estas reglas hacen parte del prototipo que se construyó en C# para generar, a partir un discurso expresado en Esquemas Preconceptuales, los diagramas de clases, comunicación y máquina de estados, tal y como se ejemplifica en la Sección siguiente.

Caso de estudio

El prototipo que se construyó para aplicar las reglas de la sección anterior, utiliza como entrada un Esquema Preconceptual elaborado en Microsoft Visio™, con una plantilla previamente definida, y arroja como resultado un archivo de texto con los elementos identificados para cada uno de los diagramas mencionados. Como caso de estudio se presenta el Esquema Preconceptual de una biblioteca ficticia, el cual se muestra en la figura 2. La figura 3 muestra una interfaz gráfica de usuario del prototipo.

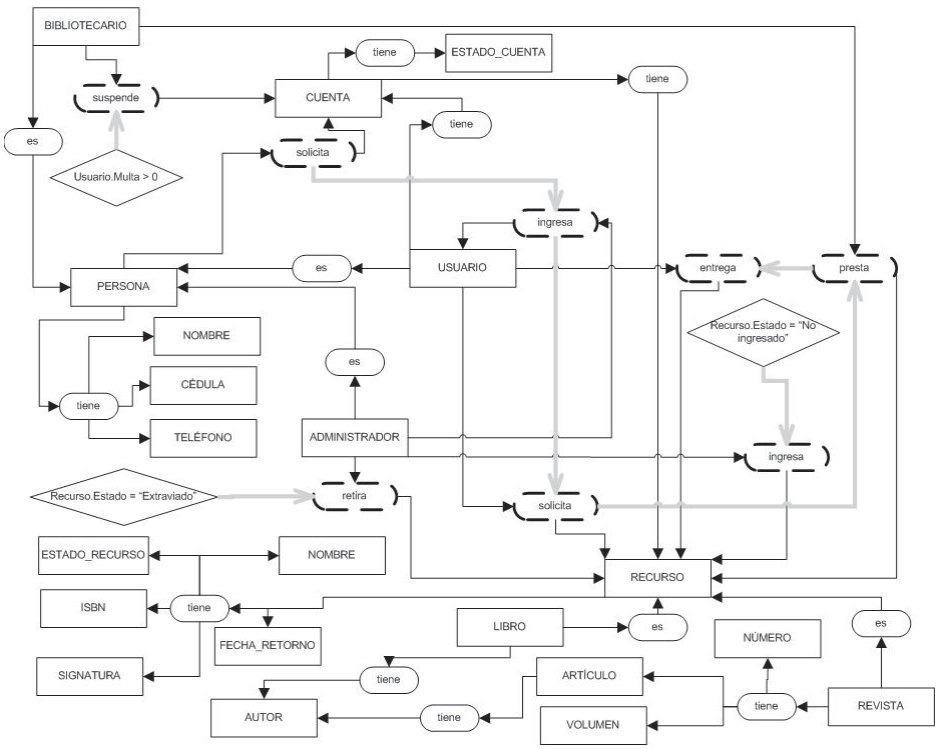

Figura 2 Esquema Preconceptual de una biblioteca ficticia

Una vez se ha cargado el Esquema Preconceptual de la Figura 2, el prototipo arroja como resultado un archivo de texto, cuyo contenido es el siguiente:

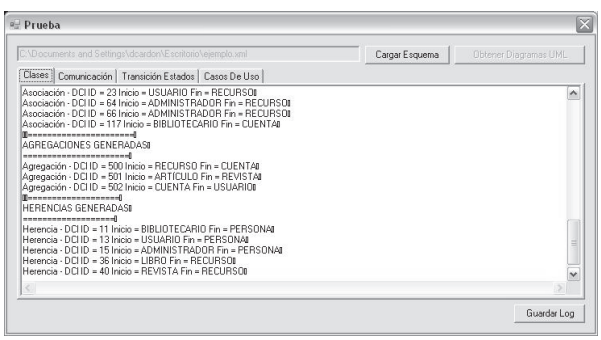

Figura 3 Interfaz gráfica de usuario del prototipo para generar los diagramas de clases, comunicación y máquina de estados a partir de un Esquema Preconceptual dado

```
==========================================
LOG DE CONVERSIÓN - DIAGRAMA DE CLASES
==========================================
==================
CLASES GENERADAS
==================
Clase - DCl ID = 4 Nombre = PERSONA
  - NOMBRE
  - TELÉFONO
  - CÉDULA
Clase - DCl ID = 31 Nombre = REVISTA
  - VOLUMEN
  - NÚMERO
Clase - DCl ID = 29 Nombre = RECURSO
  - ESTADO_RECURSO
  - SIGNATURA
  - ISBN
  - NOMBRE
  - FECHA_RETORNO
  + ingresa( )
  + solicita( )
  + retira( )
Clase - DCl ID = 32 Nombre = ARTÍCULO
  - AUTOR
Clase - DCl ID = 30 Nombre = LIBRO
  - AUTOR
Clase - DCl ID = 6 Nombre = USUARIO
  + ingresa( )
Clase - DCl ID = 85 Nombre = CUENTA
  - ESTADO_CUENTA
  + suspende( )
Clase - DCl ID = 5 Nombre = BIBLIOTECARIO
Clase - DCl ID = 7 Nombre = ADMINISTRADOR
========================
ASOCIACIONES GENERADAS
========================
Asociación - DCl ID = 22 Inicio = USUARIO Fin = RECURSO
Asociación - DCl ID = 70 Inicio = ADMINISTRADOR Fin = USUARIO
Asociación - DCl ID = 23 Inicio = USUARIO Fin = RECURSO
Asociación - DCl ID = 64 Inicio = ADMINISTRADOR Fin = RECURSO
Asociación - DCl ID = 66 Inicio = ADMINISTRADOR Fin = RECURSO
========================
AGREGACIONES GENERADAS
========================
Agregación - DCl ID = 500 Inicio = ARTÍCULO Fin = REVISTA
Agregación - DCl ID = 501 Inicio = CUENTA Fin = USUARIO
======================
```

HERENCIAS GENERADAS
==========
Herencia - DCl ID = 11 Inicio = BIBLIOTECARIO Fin = PERSONA
Herencia - DCl ID = 13 Inicio = USUARIO Fin = PERSONA
Herencia - DCl ID = 15 Inicio = ADMINISTRADOR Fin = PERSONA
Herencia - DCl ID = 36 Inicio = LIBRO Fin = RECURSO
Herencia - DCl ID = 40 Inicio = REVISTA Fin = RECURSO
==========
LOG DE CONVERSIÓN - DIAGRAMAS DE COMUNICACIÓN
==========
==========
DIAGRAMA #1
==========
==========
OBJETOS GENERADOS
==========
Objeto - DC ID = 17 Nombre = ADMINISTRADOR
Objeto - DC ID = 33 Nombre = RECURSO
==========
COMUNICACIONES GENERADAS
==========
Comunicacion - DC ID = 65 Mensaje = 1: RECURSO.ESTADO = “EXTRAVIADO” / retira() >> Inicio = ADMINISTRADOR Fin = RECURSO
==========
DIAGRAMA #2
==========
==========
OBJETOS GENERADOS
==========
Objeto - DC ID = 18 Nombre = ADMINISTRADOR
Objeto - DC ID = 34 Nombre = RECURSO
==========
COMUNICACIONES GENERADAS
==========
Comunicacion - DC ID = 19 Mensaje = 1: RECURSO.ESTADO = “NO INGRESADO” / ingresa() >> Inicio = ADMINISTRADOR Fin = RECURSO
==========
DIAGRAMA #3
==========
==========
OBJETOS GENERADOS
==========
Objeto - DC ID = 20 Nombre = ADMINISTRADOR
Objeto - DC ID = 24 Nombre = USUARIO
Objeto - DC ID = 35 Nombre = RECURSO
==========

```
COMUNICACIONES GENERADAS
==========================
Comunicacion - DC ID = 69 Mensaje = 1: ingresa() >> Inicio = ADMINISTRADOR Fin = USUA-
RIO
Comunicacion - DC ID = 25 Mensaje = 2: solicita() >> Inicio = USUARIO Fin = RECURSO
=====================================================
LOG DE CONVERSIÓN - DIAGRAMA DE MÁQUINA DE ESTADOS
=====================================================
==================
DME'S GENERADOS
==================
=============
DME RECURSO
=============
===================
ESTADOS GENERADOS
===================
- Estado - DTE ID = 63 Nombre estado = RETIRADO
- Estado - DTE ID = 8 Nombre estado = INGRESADO
- Estado - DTE ID = 21 Nombre estado = SOLICITADO
========================
TRANSICIONES GENERADAS
========================
- Cambio de Estado - DTE ID = 6300 Precondiciones = Recurso.Estado = "Extraviado" Inicio =  Fin
= RETIRADO
- Cambio de Estado - DTE ID = 200 Precondiciones = Recurso.Estado = "No ingresado" Inicio =  Fin
= INGRESADO
- Cambio de Estado - DTE ID = 2100 Precondiciones =  Inicio =  Fin = SOLICITADO
=============
DME USUARIO
=============
===================
ESTADOS GENERADOS
===================
- Estado - DTE ID = 68 Nombre estado = INGRESADO
========================
TRANSICIONES GENERADAS
========================
- Cambio de Estado - DTE ID = 6800 Precondiciones =  Inicio =  Fin = INGRESADO
```

Nótese que, además del diagrama de clases, se generaron tres (3) diagramas de comunicación y dos (2) diagramas de máquina de estados, correspondientes estos últimos a los conceptos "Recurso" y "Usuario". El prototipo identifica las reglas que puede aplicar de acuerdo con las diferentes triadas que logra identificar en el Esquema Preconceptual; por ejemplo, para la triada "persona tiene nombre" selecciona la regla 4.1.2 y para "usuario solicita recurso" aplica la regla 4.1.5. Para esta última triada, como también pertenece a una implicación, aplica otra de las reglas que no se describieron en este artículo, para identificar "solicita" como un mensaje del diagrama de

comunicación, y luego aplica la regla 4.3.1 para definir el estado "solicitado", perteneciente al diagrama de máquina de estados de "Recurso".

Se debe aclarar que el prototipo genera siempre el mismo conjunto de diagramas de UML para un Esquema Preconceptual dado. Sin embargo, pueden existir muchos Esquemas Preconceptuales que representen el mismo discurso en lenguaje natural. Por ser una forma de lenguaje controlado, los Esquemas Preconceptuales requieren un procesamiento previo, que corre a cargo de los analistas, pero en el cual los interesados en el desarrollo de la aplicación de software pueden participar activamente; desde el punto de vista del analista, el conocimiento de las reglas de conversión le puede suministrar herramientas de comunicación con el interesado; desde la perspectiva del interesado, las sugerencias que le haga el analista en términos de los Esquemas Preconceptuales pueden ser claramente entendidas, y pueden suministrar pistas para corregir frases que puedan presentar ambigüedad en el discurso. De esta manera, el prototipo implementado contribuye a mejorar la comunicación entre analistas e interesados, con el fin de mejorar la calidad de los diagramas de UML resultantes.

Conclusiones

En este artículo se presentó un prototipo elaborado en C# que permite la obtención automática de tres diagramas UML 2.0 (clases, comunicación y máquina de estados) a partir de un discurso representado mediante los denominados Esquemas Preconceptuales. Las contribuciones principales de este trabajo son:

- El interesado no tiene que conocer las reglas de conversión del Esquema Preconceptual en los diagramas de UML mencionados. Tampoco es necesario que conozca los detalles correspondientes a cada uno de los diagramas mencionados.
- Al ser un proceso automático se evitan los errores humanos en la aplicación de las reglas. En este sentido, el prototipo supera los trabajos previos que requerían asistencia por parte del analista en la generación de los diagramas, los que presentaban reglas ambiguas y los que no estaban implementados.
- Si bien es cierto que los diagramas generados están lejos de ser diagramas de diseño, el prototipo presentado permite obtener versiones preliminares de estos diagramas, que puedan servir como insumo para un proceso de refinamiento iterativo de ellos, con la participación activa de los interesados.
- Este prototipo puede servirle al analista para estimar cuánta información relacionada con el dominio debe recolectar, por ejemplo, si obtiene muchas clases sin atributos, el analista intuirá que falta información del objeto del negocio correspondiente a esa clase. Esta información podrá ser recolectada de manera sencilla, haciéndole las preguntas correctas al interesado en un lenguaje entendible para él. Por ejemplo, en el caso de estudio, el analista podría preguntar "¿Es cierto que un libro tiene ISBN?" y de esta manera estaría expresando la triada "libro tiene ISBN" en forma interrogativa, buscando su validación por parte del interesado.
- Los diagramas obtenidos son consistentes entre sí. Por ejemplo, se puede notar en el caso de estudio que existe un estado "solicitado" en el diagrama de máquina de estados correspondiente a "Recurso", el cual posee un mensaje llamado "solicita()" correspondiente a una comunicación que llega a la clase de objeto "Recurso" en el diagrama de comunicación y, a su vez, posee una operación "solicita()" perteneciente a la clase "Recurso" en el diagrama de clases. En el caso de que el analista estuviera creando cada uno de los diagramas de forma manual o asistida por una herramienta que no realice este tipo de análisis, habría una alta probabilidad de que estas reglas de consistencia no se tuvieran en cuenta al generar los diagramas.

Trabajo futuro

Las líneas de trabajo futuro que se pueden desprender del análisis realizado en este artículo son:

- La integración del prototipo presentado con un entorno gráfico que permita una fácil edición tanto de los Esquemas Preconceptuales como de los diagramas UML resultantes.
- La definición e implementación de un conjunto de reglas que le muestren al interesado problemas potenciales de completitud o consistencia, por ejemplo el hecho de que una relación dinámica estuviera fuera de un flujo de implicaciones (lo cual genera errores al momento de generar los diagramas).
- La identificación de información presente en el Esquema Preconceptual relacionada con otros diagramas UML 2.0 como el diagrama de casos de uso o el diagrama de actividades.
- El incremento en la capacidad expresiva del Esquema Preconceptual para expresar interacciones entre relaciones dinámicas diferentes a la implicación.
- La elaboración de una aplicación que permita obtener automáticamente un Esquema Preconceptual a partir de una serie de frases en UN-Lencep [10].

Referencias

1. OMG. "OMG Unified Modeling Language Specification". *Object Management Group.* http://www.omg.org/UML/. Consultada el 10 de Junio de 2007.
2. P. Chen. "English Sentence Structure and Entity-Relationship Diagram". *Information Sciences*. Vol. 1. 1983. pp. 127–149.
3. A. Gangopadhyay. "Conceptual modeling from natural language functional specifications". *Artificial Intelligence in Engineering.* Vol. 15. 2001. pp. 207–218.
4. B. R. Bryant. "Object-Oriented Natural Language Requirements Specification". *23rd Austalasian Computer Science Conference.* Canberra. 2000. pp. 24–30.
5. S. Konrad, B. H. C. Cheng. "Automated Analysis of Natural Language Properties for UML Models". *Satellite Events at MoDELS 2005 Conference, Lecture Notes in Computer Science.* N.° 3844. 2006. pp. 48–57.
6. N. Juristo, J. Morant, A. Moreno. "A formal approach for generating OO specifications from natural language". *The Journal of Systems and Software*. Vol. 48. 1999. pp. 139–153.
7. G. Fliedl, Ch. Kop, H. Mayr, W. Mayerthaler, Ch. Winkler, "Linguistically Based Requirements Engineering: The Niba-Project", *Data Knowledge Engineering,* Vol. 35. 1999. pp. 111–120.
8. C. M. Zapata, A. Gelbukh, F. Arango. "Pre-conceptual Schema: A Conceptual-Graph-Like Knowledge Representation for Requirements Elicitation". *Lecture Notes in Computer Science*. Vol. 4293. 2006. pp. 17–27.
9. A. vanWijngaarden. "Orthogonal design and description of a formal language". *Technical report.* Mathematisch Centrum. Amsterdam. 1965. pp. 1–25.
10. C. M. Zapata, A. Gelbukh, F. Arango. "UN-Lencep: Obtención Automática de Diagramas UML a partir de un Lenguaje Controlado". A. Hernández, J. L. Zechinelli (Eds.) *Avances en la Ciencia de la Computación.* Guanajuato, Mexican Society of Computer Science, 2006. pp. 254–259.
11. C. M. Zapata, F. Arango, A. Gelbukh. "Pre-conceptual Schema: a UML isomorphism for automatic obtaining of UML conceptual schemas". *Research in Computing Science: Advances in Computer Science and Engineering*. Vol. 19. 2006. pp. 3–13.
12. M. Fowler. *UML Distilled*. 3th ed. Boston. Pearson Education. 2004. pp. 1–112.