Naturalistic Programming: Model and Implementation

O. Pulido, and U. Juárez

***Abstract*—Naturalistic programming is defined as a programming technique that uses abstractions whose expressiveness is close to natural languages. The objective is preserving as much as possible the needs of the client in their language, while the text of these needs is simultaneously the requirements specification and the program source code. Consequently, the goal of the naturalistic paradigm is reducing the gap between problem domain and solution domain. In the literature, two main approaches are reported, one focuses on transforming controlled natural languages into high level code, such as Java and Python; in the other approach the requirements description is at the same time the program source code. While the translators employed in the first approach do not offer a new paradigm, the few naturalistic languages reported have utility in specific domains. In the absence of a naturalistic framework, this article presents the minimum elements for defining a naturalistic model that allows the creation of general-purpose languages and at the same time, the SN language is introduced as a proof-of-concept, which is a prototype language for naturalistic programming.**

***Index Terms*—Naturalistic programming, Controlled natural english, Expressiveness, Automatic source code generation**

I. Introducción

DESDE la década de 1960 se propuso el idioma inglés como lenguaje de programación para reducir la brecha entre el dominio del problema (necesidades del cliente) y el dominio de la solución (requisitos del sistema) [1]. Sin embargo, la principal limitante es que un lenguaje natural es inherentemente ambiguo y dependiente del contexto, problema que se resuelve entre las personas gracias a su razonamiento [2]. Las computadoras procesan sin problema un lenguaje de programación porque se basan en formalismos y no se presenta ambigüedad, con la implicación de transformar los requisitos para adaptarlos tanto al lenguaje como al paradigma.

En el contexto de un paradigma, el lenguaje de programación es altamente influyente en el proceso de transformación de requisitos a código lo que resulta en la pérdida de expresividad. La expresividad es la capacidad de comunicar y representar ideas en un texto, por lo que es deseable que un lenguaje tenga alto poder expresivo para representar dichas ideas lo más fielmente posible. También, un paradigma tiene como fundamento un modelo conceptual para describir cómo se relacionan los conceptos de un problema y representarlos mediante algún artefacto adecuado [3].

Como alternativa a la ambigüedad del lenguaje natural se propusieron dos enfoques [4] 1) un conjunto formalizado y restringido del lenguaje natural y 2) el uso de heurísticas en combinación con técnicas de inteligencia artificial. El primer enfoque describe a la programación naturalística donde un lenguaje natural controlado hace referencia tanto a abstracciones como a la propia estructura del lenguaje y sus instrucciones. Por lo tanto, la programación naturalística ofrece mecanismos para desarrollar software utilizando elementos de lenguaje natural y formal junto con una gramática expresiva. Las implementaciones de lenguajes naturalísticos reportados se enfocan en problemas particulares como la automatización en el desarrollo de software [5] y en problemas matemáticos simples [6], por lo que los investigadores no se han enfocado en establecer las bases necesarias para el desarrollo formal de lenguajes naturalísticos.

Con base en la revisión exhaustiva de literatura [7], este artículo presenta un modelo conceptual para el diseño e implementación de lenguajes de programación naturalísticos junto con un lenguaje prototipo llamado SN. El modelo considera el idioma inglés y describe sustantivos, verbos, adjetivos, referencias indirectas, plurales para colecciones de datos y eventos que describen el contexto de una oración denominados circunstancias. En SN las instrucciones se definen de tres formas: sujeto-verbo-objeto, verbo-objeto y verbo-objeto-proposición-objeto. Debido a que en los lenguajes naturales hay especial énfasis en los tipos [8] se ofrece en SN la capacidad de trabajar con referencias indirectas sin depender de identificadores.

El presente artículo se estructura de la siguiente manera: en la sección 2 se presentan los antecedentes; en la sección 3 se describe el modelo propuesto; en la sección 4 se describe un extracto de la gramática de SN; en la sección 5 se presenta un escenario que permite realizar una consulta a una base de datos; en la sección 6 se discuten los resultados estadísticos de las pruebas que se hicieron con usuarios; en la sección 7 se mencionan los trabajos relacionados más relevantes; por último, en la sección 8 se muestran las conclusiones y el trabajo a futuro

II. Antecedentes

El desarrollo de software evolucionó a partir de un sistema binario, pasando por lenguajes ensambladores, hasta llegar a los paradigmas de alto nivel como la programación orientada a objetos y la programación funcional [9]. Por otro lado, los lenguajes de programación se crean pensando en mantener formalismos matemáticos que eliminen las definiciones ambiguas, lo que permite que sean procesables sin la necesidad de recursos de hardware excesivos. La formalidad de los lenguajes de programación contrasta con los lenguajes naturales, que son el medio que los clientes emplean para describir los

Oscar Pulido Prieto – Tecnológico Nacional de México/I.T. Orizaba. (opulidop@ito-depi.edu.mx).

Ulises Juárez Martínez – Tecnológico Nacional de México/I.T. Orizaba. (ujuarezm@orizaba.tecnm.mx).

requisitos de software, pero que a su vez son ambiguos y poco adecuados para la programación[1]. Lo anterior genera una brecha entre la naturalidad con la que se expresa el cliente y la formalidad con la que se programa el sistema, lo que conlleva a un proceso de ingeniería para analizar los requisitos y facilitar su transformación en código ejecutable.

A. Planteamiento del Problema

Los lenguajes de programación naturalísticos que se reportan se enfocan a que el programador defina instrucciones por medio del lenguaje natural, pero no reportan qué elementos se consideran necesarios para diseñar otros lenguajes naturalísticos porque dada la complejidad de los lenguajes naturales, los autores restringen sus trabajos a resolver problemas de dominios particulares. Dichas restricciones son a nivel gramatical por medio de una sintaxis limitada, como en el caso de NLC [11], o de análisis al definir un contexto particular para el texto, como en el caso de NLCI [12].

Los lenguajes naturalísticos que se enfocan a dominios particulares resuelven los problemas gracias a que limitan su alcance al dominio correspondiente, de modo que eliminan la ambigüedad de las oraciones gracias a su enfoque en un contexto particular. Con base en esto, establecer un contexto para las abstracciones al momento de programarlas permite a los desarrolladores crear código sin importar el dominio del problema. Para lograr lo anterior se requiere de instrucciones flexibles, donde los sustantivos, verbos, adjetivos, sintagmas e inclusive al contexto particular de cada oración se tomen en cuenta, de modo que un lenguaje naturalístico de propósito general consiste en instrucciones flexibles, pero controladas donde la ambigüedad se elimine.

Los lenguajes naturales poseen la limitante de ser poco descriptivos cuando se trata de conceptos formales, algo que descartaron los autores de los lenguajes naturalísticos que se reportan debido a que sus lenguajes y harramientas se centran en dominios particulares, de modo que la gramática mantiene el soporte para los formalismos del dominio. Por el contrario, se espera que un lenguaje naturalístico de propósito general permita integrar no sólo las propiedades de los lenguajes naturales, sino también los formalismos de cualquier dominio. Por tanto, se requiere de un mecanismo que permita establecer dichos formalismos sin reducir la expresividad de los lenguajes de programación respecto al lenguaje natural, de modo que el código se estructure de forma similar a los documentos científicos, donde los formalismos se separan de forma clara, ya sea cambiando el tipo de letra o en un espacio separado.

B. Modelos de Programación

El modelo funcional posee los elementos mínimos para la implementación de una máquina universal de Turing [13]. Al agregar determinadas características a este modelo se obtienen otros modelos de programación. Por ejemplo, agregar la capacidad para trabajar con valores únicamente dentro de un bloque da como resultado encapsulación y por tanto, instancias y abstracción, que son elementos que conforman al paradigma orientado a objetos.

Por otro lado, en [9], se describe al modelo de objetos como un conjunto de cuatro elementos fundamentales que son la abstracción, la encapsulación, la modularidad y la jerarquía. Donde la abstracción consiste en tomar los elementos necesarios para una aplicación, ignorando los detalles irrelevantes. La encapsulación es el proceso de ocultar los elementos internos de una abstracción y para que se acceda a ellos de forma controlada. La modularidad trata sobre agrupar dichas abstracciones con base en su funcionalidad (o contexto). Por último, la jerarquía permite clasificar las abstracciones ya sea como subtipos o como elementos que conforman a otra abstracción.

Una limitante con los objetos es la falta de soporte para encapsular eficientemente los requerimientos no funcionales, mismos que se encuentran dispersos entre abstracciones. En [14] los autores describen la programación orientada a aspectos como un paradigma complementario a los objetos en forma de un modelo de puntos de unión. El modelo de puntos de unión se conforma de tres elementos: el *punto de unión* (*join point*) es evento identificable en el código (una instanciación, una asignación o un método) donde es posible agregar funcionalidad adicional. El *corte* (*pointcut*) es un mecanismo para especificar y cuantificar puntos de unión. Por último, el *aviso* (*advice*) consiste en el código adicional que se agregará al punto de unión con base en la especificación del corte.

Los aspectos presentan una mejoría en la encapsulación de requisitos dispersos, pero todavía mantienen la expresividad de los lenguajes de programación tradicionales. En [15], los autores discuten acerca de cómo elementos de los lenguajes naturales se podrían agregar a los lenguajes formales para incrementar la expresividad de los lenguajes de programación, de modo que su sintaxis sea más cercana al inglés, pero sin perder su formalidad. En particular, los autores mencionan cómo la capacidad para utilizar referencias estructurales, temporales y reflexivas de lenguaje natural elevaría el nivel de expresividad de los lenguajes de programación. Por ejemplo, las referencias estructurales son comunes en la programación, pero las referencias temporales no. *AspectJ* utiliza una forma rudimentaria de referencias temporales, pero es incapaz de describir información del contexto, donde operaciones como *the last operation* podría ayudar a definir mecanismos más cercanos al lenguaje natural en términos de expresividad.

C. Programación Naturalística

La programación naturalística es un paradigma que se basa en la integración de elementos de los lenguajes naturales para el diseño de lenguajes de programación con una sintaxis más cercana al lenguaje natural. En [15] se abordan las ventajas y limitantes tanto de los lenguajes naturales, como de los lenguajes de programación actuales en el desarrollo de software. También se menciona que los lenguajes naturales son poco útiles dada su ambigüedad, mientras que los lenguajes de programación son difíciles de mantener si el código carece de la documentación adecuada, esto se debe a que la gramática

[1]En [10] se presenta una revisión del uso de lenguaje natural en las etapas de especificación y análisis de requisitos por parte de diversos autores.

de los lenguajes de programación fuerza a los programadores a transformar y descomponer los requisitos para adaptarlos a la sintaxis del lenguaje de programación. Como solución los autores proponen un término medio; es decir, un lenguaje de programación formal que posea elementos de los lenguajes naturales tales como las referencias anafóricas.

D. Trabajos Relacionados

En [16] se presenta *Pegasus*, una herramienta que permite el desarrollo de software a partir de una entrada en lenguaje natural que se basa en un mecanismo que asemeja al cerebro humano, depende de un diccionario de entidades que asocia conceptos que se definen con palabras y sus conjugaciones. Pegasus es independiente del lenguaje del programador, de modo que una entrada en inglés o español produce el mismo código en Java, C# o Python.

En [17], [18], [19], [20] se presenta *Attempto Controlled English* (ACE), una versión formalizada del idioma inglés que es transformable a lógica de primer orden. Aún y cuando es un claro ejemplo de un lenguaje natural formalizado, dado que ACE no se pensó como medio para programar, no ofrece un mecanismo para dicha tarea.

En [21], [22] se presenta el lenguaje *Computer-Processable Language* (CPL), que es un lenguaje natural controlado que se desarrolló bajo el enfoque naturalístico, pero que además posee de forma embebida un lenguaje formal denominado *CPL-Lite*. CPL se basa en heurísticas para resolver la ambigüedad de un texto y de ese modo producir la salida que el programador espera, en caso de no hacerlo, el programador emplea CPL-Lite para producir una salida libre de ambigüedades dado que su enfoque formal carece de la misma.

En [6] se presenta *Metafor*, una herramienta que se pensó como apoyo a los programadores novatos, permite describir un problema en lenguaje natural y devuelve como resultado un esqueleto en lenguaje Python; además, se plantea la posibilidad de que programadores experimentados utilicen Metafor para generar una lluvia de ideas que los ayuden en etapas tempranas del desarrollo de software. Metafor trabaja a través de un "diálogo" con la computadora que refina la estructura de clases que genera durante sucesivas descripciones del programador.

En [5], los autores presentan *Natural Language Computer* (NLC), un lenguaje de programación que se enfoca al manejo de arrays que aparecen en pantalla por medio de un subconjunto del idioma inglés para resolver los problemas de expresividad del lenguaje C. Lo destacable de su gramática es la forma en cómo analiza oraciones simples para trabajar en un esquema naturalístico. Se observa el uso de referencias indirectas y el uso de preposiciones y artículos para el manejo de instrucciones simples.

En [23], se presenta *ReDSeeDS*, una herramienta que toma una especificación de requisitos en lenguaje natural sin importar la lengua por medio de una versión enriquecida del modelo semántico. Los autores mencionan que dicho enriquecimiento se logró por medio de patrones en XML que presentan la información de los escenarios de prueba, de modo que se eliminan las ambigüedades en el significado y de esta forma se genera código Java.

En [24], se presenta la herramienta *MOOSE Crossing*, un *MUD* (*Multi-User Dungeon*) educativo para niños, los MUD son juegos de rol multi-usuario basados en texto. MOOSE Crossing se basa en los MOO (*MUD, Object-Oriented*), que son una variante de los MUD que permiten modificar el servidor por medio de programación orientada a objetos. En [25], se obtuvo que MOOSE Crossing requiere de una instrucción previa, ya que en caso contrario los niños utilizarán contrucciones gramaticales inválidas.

III. Modelo Naturalístico

Con base en la revisión que se reporta en [7], se observó lo que los autores de dichos trabajos consideran al momento de diseñar lenguajes de programación con propiedades naturalísticas. Los trabajos se enfocaron en resolver problemas de un dominio particular por medio de sustantivos y adjetivos, dejando en segundo término los pronombres, esto se debe a que su uso depende del contexto y generan ambigüedad. Varios autores manejaron únicamente sintagmas nominales (frase cuyo núcleo es el sustantivo) para complementar sustantivos con adjetivos, pero no como esquemas de categorización o filtrado de entidades. También se observó que en los lenguajes naturales la comunicación se realiza principalmente por medio de referencias indirectas (ya sean pronombres o sintagmas nominales) y que los identificadores se reservan para situaciones muy particulares, como nombres propios o cosas irrepetibles, como fechas o conceptos abstractos.

Con base en el estudio que se realizó, que consistió en identificar los elementos del lenguaje natural que los autores de los trabajos que se reportan en [7] consideraron para sus lenguajes y herramientas de programación, se llegó a la conclusión de que los elementos mínimos que se requieren para diseñar un lenguaje naturalístico de propósito general, con expresividad semejante al idioma inglés y con la formalidad mínima necesaria para que una computadora lo procese sin ambigüedad, son los siguientes:

1 Sustantivo como abstracción base que es singular o plural.
2 Adjetivo como complemento al sustantivo.
3 Verbo como acción que realiza un sustantivo.
4 Circunstancia como condicionante que responde a eventos.
5 Sintagma para definir instrucciones con una complejidad más allá del sujeto y el predicado.
6 Anáfora como mecanismo que permite definir instrucciones en términos de elementos que se describieron con anterioridad.
7 Definición explícita de tipos, de modo que se utilicen en la construcción de instrucciones que se conformen con sintagmas.
8 Expresividad similar a una formalización del inglés para evitar ambigüedad.

El uso de adjetivos y sustantivos para definir mecanismos de abstracción y especialización permite establecer sintagmas donde un sustantivo posee opcionalmente uno o más comportamientos especializados dependiendo si se combina o no con adjetivos. La definición de verbos con una sintaxis flexible

permite establecer instrucciones en términos de sintagmas verbales, que básicamente son el predicado de una oración. Dependiendo de la complejidad del lenguaje, un lenguaje basado en el modelo permite o no *sujeto elíptico*. Por último, las circunstancias permiten definir un contexto para un sintagma, de modo que el uso de circunstancias sirva para establecer condicioness para la composición de sustantivos (exclusión mutua de adjetivos, necesidad o negación de un adjetivo para un sustantivo particular). Además, las circunstancias permiten definir un contexto de ejecución para una oración, donde un verbo se ejecuta como un contexto previo o posterior de otra oración.

Además, hay elementos que no son esenciales, pero que se consideran deseables ya que ofrecen un nivel de expresividad aún mayor, dichos elementos se presentan a continuación:

1 Deixis completa, lo que implica que las instrucciones se definan en función de elementos que se describan no sólo antes, sino también después en el mismo texto.
2 Identificadores que permitan trabajar con las abstracciones por medio de referencias directas.
3 Tipificación basada en propiedades, lo que implica clasificar un sustantivo con base en sus propiedades y por tanto, que se tenga la capacidad para agregar nuevas propiedades a una abstracción cuando se cree.

El manejo completo de la deixis permite que un lenguaje posea mayor capacidad para trabajar con referencias indirectas estructurales, debido a que el uso de catáforas permite indagar sobre las instrucciones subsecuentes. El uso de identificadores agrega un soporte gramatical para destacar instancias particulares en un conjunto, lo que facilita su manejo y permite describir instrucciones más concisas. Por último, la tipificación basada en propiedades permite agregar nuevos elementos a un sustantivo, lo que facilita la extensibilidad del mismo al momento de la creación de una instancia, de modo que se requiere conocer las propiedades del sustantivo para una identificación más exacta. Por ejemplo, en el contexto de una inmobiliaria, particularmente en una lista de las casas que se ofrecen, la frase *la casa con sótano* permite identificar a una casa con base en una de sus características, en este caso el que posea un sótano, de modo que este atributo particular permite descartar a las casas que carecen de sótano, pero además agrupar a todas aquellas casas que lo posean.

IV. Lenguaje SN

En esta sección se presenta un extracto de la gramática del prototipo de lenguaje naturalístico de propósito general SN, mismo que se construyó con base en el modelo que se describe en la sección anterior. La gramática de SN es un subconjunto formalizado del idioma inglés y genera *bytecode* de Java. SN utiliza referencias indirectas que se basan en los pronombres *it*, *these* y *this*; permite realizar composición entre sustantivos y adjetivos para agregar un contexto; soporta una sintaxis mixta para la definición de oraciones donde la definición del verbo indica su posición en la oración (con sujeto explícito o deíctico[2]); por último, el uso del concepto de circunstancia, que permite definir un contexto con base en eventos. Ya que la idea principal es implementar el modelo, no se tomó en cuenta la optimización de código. Cabe destacar que dado lo extenso de la implementación, se describirán únicamente los elementos más relevantes de la gramática[3].

En el lenguaje SN, las palabras reservadas se escriben con minúscula, para sustantivos y adjetivos se recomienda utilizar la notación conocida como *upper camel case*, mientras que para atributos y verbos se recomienda la notación conocida como *lower camel case* (aunque no hay problema en el uso de mayúsculas y minúsculas siempre y cuando no se utilicen palabras reservadas como sustantivos, adjetivos, verbos o identificadores). Se recomienda de este modo para crear una correcta distinción entre las abstracciones principales y el resto de las palabras. Al basarse en la Máquina Virtual de Java (JVM), SN genera *bytecode* de Java, de modo que las abstracciones en la correspondiente implementación hacen referencia a sustantivos y adjetivos que equivalen a clases e interfaces respectivamente.

A. Oraciones

Dado que una oración se compone de sujeto y predicado, se requiere de un verbo que complemente al sustantivo indicando qué acciones realiza y si alteran o no su estado y el de otros sustantivos. Generalmente los programadores se expresan en función de construcciones donde se espera que quien ejecute la acción (desde una perspectiva gramatical) sea la computadora. Se propone un mecanismo que permite indicar en qué posición de la instrucción que llama al verbo se encuentra el sustantivo que lo contiene, para lo cual se emplea la palabra reservada *itself*. A continuación se presentan las reglas que se encargan de describir esta clase de instrucciones:

```
vd ::= '\t' vs ':' i | '\t' vs '.' '\n'
vs ::= 'verb' ID 'itself'
```

Cuyo ejemplo de implementación en el lenguaje SN es el siguiente:

```
noun House:
  attribute availability is true.
  verb sell itself:
    available is false.
```

Como se observa, se emplea la palabra reservada *verb*, seguida del nombre del verbo. Por medio de esta notación se define la posición que ocupa el sustantivo cuando se utiliza en una instrucción, en este caso se indica que el sustantivo actúa como objeto directo del verbo:

```
a House.
sell the House.
```

La oración *sell the House* indica una instrucción donde el verbo es *sell* y *the House* es quien lo utiliza, mismo que en la firma se define como *itself*.

Dado que la referencia al propio sustantivo se encuentra en la firma, la misma varía su posición dependiendo del tipo de oración que se requiere formar, esto se logra al cambiar de posición para crear firmas con una complejidad diferente.

[2]Se dice que el sujeto de una oración es deíctico cuando se omite y sólo aparece el predicado, de modo que su contexto se infiere.

[3]La gramática se encuentra en el anexo disponible en la siguiente dirección: https://bit.ly/2UlZBwQ.

En este caso, se presenta una definición de verbo donde el sustantivo que lo define actúa como *objeto indirecto*, mientras que el argumento es un *objeto directo*. Con base en lo anterior, la regla vs se observa de la siguiente forma:

```
vs  ::= 'verb' ID 'itself' | 'verb' ID p 'itself'
     |  'verb' ID a p 'itself'
arg ::= ID 'as' np | ac 'and' ID 'as' np
ac  ::= ID 'as' np | ac ',' ID 'as' np
```

Un ejemplo de implementación se presenta a continuación:

```
1 noun House:
2   attribute owner as Client.
3   verb add buyer as Client to itself:
4     the owner of this is buyer.
```

Cuya invocación se muestra en el siguiente ejemplo:

```
a House.
a Client.
add the Client to the House.
```

Donde *add* es el verbo, *the Client* es el objeto directo y *the House* el objeto indirecto, que además es el que contiene la instrucción y por tanto, se representa en la firma con *itself*.

Por último, se provee el soporte para definir verbos donde el sustantivo que los contenga actúe como *sujeto gramatical* de forma semejante a las sintaxis de lenguajes como Java. A continuación se presenta la regla *vs* integrando esta clase de construcciones:

```
vs ::= 'verb' ID 'itself' | 'verb' ID p 'itself'
    | 'verb' ID arg p 'itself'
    | 'verb' 'itself' ID arg
    | 'verb' ID arg
```

Cabe destacar el último caso, ya que la construcción *'verb' ID arg* por defecto se considera como *'verb' 'itself' ID arg*. A continuación se presenta un ejemplo de su utilización:

```
noun House:
  attribute availability is true.
  verb itself sold:
    availability is false.
```

Que se utiliza de la siguiente forma:

```
a House.
the House sold.
```

Donde, el verbo *sold* es el predicado de la oración, mientras que *the House* es el sujeto gramatical, que además contiene al verbo y en la firma se representa con *itself*.

V. Escenario: Conexión Naturalística a Bases de Datos

A continuación se presenta un escenario de prueba donde se describe un mecanismo naturalístico para realizar una conexión a una base de datos, un sustantivo se combina con un adjetivo que le permite actuar como una entidad de base de datos, de este modo se desacopla la conexión del sustantivo y se crea un mecanismo genérico para realizar dicha tarea. Este escenario se consideró para verificar la correctitud y expresividad del código. De este modo, se observó si las abstracciones que se establecieron son expresivas desde la perspectiva del lenguaje natural, al tiempo que realiza la tarea que se infiere de su lectura. A continuación se presenta un adjetivo *Storable*, que refina al adjetivo *DBEntity*.

```
1 adjective Storable is DBEntity:
2   circumstance: execute assign this when
     this is created.
3   verb assign itself:
4     the driver is "org.postgresql.Driver".
5     the jar is "C:\\pgsql.jar".
6     the user is "admin".
7     the password is "admin1".
8     the URL is "jdbc:postgresql:
       //localhost:5432/agenda".
```

En la línea 1, se establece el nombre del adjetivo *Storable*, que es un refinamiento del adjetivo *DBEntity*, mismo que se encarga de gestionar la conexión y el acceso a bases de datos. En la línea 2 se define una circunstancia, que es un mecanismo para definir el contexto de un verbo o un sustantivo, en este caso indica que el verbo *assign* se ejecuta cuando se crea una instancia que se complemente con el adjetivo *Storable*. La línea 3 es la firma de un verbo, donde *itself* especifica qué posición tomará la instancia que lo invoque, en este caso describe una oración con sujeto elíptico donde el invocador funciona como complemento directo (*assign the Storable.*). De las líneas 4 a 8 se cargan los elementos que se requieren para acceder a una base de datos, en este caso se utilizó el gestor de bases de datos *PostgreSQL*.

Con el refinamiento anterior, se requiere de un sustantivo que implemente dicho mecanismo, para este ejemplo se utilizará el sustantivo *Person*.

```
noun Person:
  attribute name as a String.
  attribute age as an Integer Number.
```

Con base en lo anterior, la combinación entre *Person* y *Storable* permite trabajar con bases de datos, al tiempo que se mantiene una separación de asuntos (*concerns*) y por tanto, el mantenimiento del código es más sencillo de realizar. A continuación se presenta la implementación de los ejemplos anteriores.

```
1 main Select:
2   a DBPersistent String with "John Doe" as value.
3   a Storable Person with the String as name.
4   some Strings.
5   add "name" to these.
6   add "age" to these.
7   select the Strings from the Person.
8   System prints these and newline.
```

La línea 1 describe una abstracción *main*, que es el punto de inicio de cualquier programa en SN. La línea 2 describe la instancia de un sustantivo *String* que se complementa con un adjetivo *DBPersistent*, lo que convierte a la cadena en una abstracción almacenable en un atributo de una tupla de una base de datos. La línea 3 indica la creación de una instancia del sustantivo *Person* que se complementa con *Storable*, de modo que dicha instancia funciona como un elemento que equivale a una tupla de una base de datos. La línea 4 indica la creación de un plural de cadenas *Strings*, que almacenará los atributos de la tabla a los que se desea acceder. En las líneas 5 y 6 se agregan las cadenas *name* y *age*, que son los atributos de la tabla *Person* que se desea extraer de la base de datos. En la línea 7 se realiza la consulta a la base de datos por medio del verbo *select*, que devuelve un plural del tipo *Person*, si no se define un plural, entonces se devuelve el tipo

Things, que es la raíz de los plurales. Por último, en la línea 8 se imprimen los valores[4].

VI. Evaluación del Prototipo

Como se observa en el escenario que se presenta en la sección anterior, la gramática de SN permite trabajar con referencias indirectas, además de reducir el uso de identificadores como mencionan los autores en [15], al tiempo que se mantiene una gramática similar al idioma inglés.

Cabe destacar que evaluar la expresividad desde la perspectiva del lenguaje natural es algo prácticamente imposible ya que como se menciona en [26], no se reporta un mecanismo estandarizado para evaluar la traducción entre *man pages* (documentación de software para sistemas basados en UNIX), es decir: no se reporta un mecanismo que permita evaluar de forma objetiva un programa y lo que se espera que realice (sus intenciones descritas en lenguaje natural)[5]. Además, los mecanismos que se reportan en otros trabajos que se revisaron se enfocan al código que resultó de la traducción (si es el caso) o en su defecto, a pruebas experimentales por parte de personas con diverso nivel de conocimiento en programación (desde niños, hasta arquitectos de software y programadores). Por este motivo se seleccionaron instrucciones y fragmentos de código que abarcaran la mayor cantidad de construcciones válidas de SN, para que posteriormente alumnos con diversos niveles de experiencia tanto en programación como en inglés evaluaran su expresividad desde la perspectiva del lenguaje natural.

Se eligió una evaluación por medio de análisis en lugar de una evaluación práctica para no sesgar las respuestas con un antecedente basado en experiencia con el paradigma, ya que durante las pruebas iniciales del prototipo, se observó que quienes trabajan SN sin tener experiencia con la parte naturalística, buscan plasmar ideas conforme a paradigmas que dominen. Por ejemplo, SN permite trabajar con una sintaxis más cercana al paradigma orientado a objetos, pero esto se logra a costa de sacrificar la expresividad naturalística que se espera del prototipo. Con base en lo anterior, se observa que se requiere de un proceso de "des-aprendizaje" del paradigma que se domine o, en su defecto, enfocarse a gente sin conocimientos de programación para reducir el sesgo en la evaluación de la expresividad de SN.

Las pruebas se realizaron por medio de una escala de Likert [29] para una evaluación subjetiva de la expresividad, ya que la correctitud de una oración no depende únicamente de la gramática, sino también de su contexto. Por ejemplo, la oración *Colorless green ideas sleep furiously* descrita en [30] por Noam Chomsky es gramaticalmente correcta, pero carece de sentido semántico por las contradicciones que se generan en *colorless* con *green* y en *sleep* con *furiosuly*, ya que definen ideas mutuamente excluyentes. En SN este ejemplo se implementa de la siguiente forma: *the Colorless and Green Idea sleeps furiously*, Donde *Colorless* y *Green* son adjetivos, *Idea* es un sustantivo, *sleeps* un verbo con formato *itself sleeps emotion as Emotion* y por último, *furiously* es un identificador de tipo *Emotion*.

A. Pruebas

Para las pruebas se solicitó el apoyo de alumnos de la carrera de ingeniería en sistemas computacionales del Tecnológico Nacional de México, campus Orizaba, donde 49 alumnos revisaron ejemplos de programas hechos con SN. Los alumnos se encuentran en un rango de edad de entre 18 y 32 años. El 46.9% de los alumnos consideran tener un nivel de programación medio, el 2% consideran tener un nivel alto mientras que el 51.1% consideró tener un nivel bajo o nulo. En el dominio del inglés, se observa que el 8.1% de los alumnos consideran tener un nivel alto, el 53.1% consideran tener un nivel medio, mientras que el 38.7% consideraron tener un nivel bajo o nulo.

TABLA I
Resultados de preguntas

Pregunta	1	2	3	4
1	10.2%	36.7%	40.8%	12.2%
2	2%	38.8%	51%	8.2%
3	8.2%	34.7%	36.7%	20.4%
6	10.2%	34.7%	49%	6.1%
7	2%	38.8%	46.9%	12.2%
8	4.1%	34.7%	38.8%	22.4%
9	2%	26.5%	57.1%	14.3%
10	4.1%	36.7%	46.9%	12.2%
11	4.1%	32.7%	51%	12.2%
12	10.2%	28.6%	49%	12.2%
13	6.1%	32.7%	38.8%	22.4%
14	14.3%	32.7%	40.8%	12.2%
15	14.3%	20.4%	57.1%	8.2%
16	6.1%	30.6%	40.8%	22.4%
17	8.2%	28.6%	53.1%	10.2%
18	0	8.2%	59.2%	32.7%
19	4.1%	42.9%	40.8%	12.2%
20	10.2%	22.4%	59.2%	8.2%
21	6.1%	22.4%	55.1%	16.3%
22	0	20.4%	59.2%	20.4%

TABLA II
Preguntas verdadero/falso

Pregunta	Sí	No
4	55.1%	44.9
5	51%	49

La escala se estableció con valores entre el 1 y el 4, donde 1 es *nada expresivo* y 4 es *totalmente expresivo*, en la Tabla I se presentan los resultados para estas preguntas[6]. Las pruebas consistieron en analizar desde instrucciones simples para entender su significado y contexto, hasta el análisis de programas completos que realizan tareas como operaciones aritméticas por medio de oraciones en inglés; programas que definen sustantivos, adjetivos y sus combinaciones; o el manejo de bases de datos. Por otro lado, en la Tabla II se presentan los porcentajes de respuestas dadas a dos preguntas cuyas opciones eran "sí" o "no".

[4]Para las operaciones de inserción, actualización y borrado, véase el siguiente anexo: https://bit.ly/2I1TcAm.

[5]En [27], [28], los autores proponen técnicas de inteligencia artificial como solución a largo plazo para verificar la consistencia entre el software y la documentación informal, que es en lenguaje natural.

[6]Una evaluación de cada pregunta se encuentra en el anexo disponible en la siguiente dirección: https://bit.ly/2I3o57M.

A continuación se presenta un extracto de los porcentajes que se obtuvieron del análisis que realizaron los alumnos al lenguaje SN. Pregunta 4: Con base en las siguientes instrucciones de asignación:

```
a is 25. b is 54. c is 62. d is 86. e is 35.
f is 28. g is 55. h is 33. i is 24. j is 5.
k is 2.
```

Para instrucciones de invocación con referencias indirectas tales como *System prints the last 2 Number and newline*, se observó que para la pregunta "¿qué variable (o variables) almacena el resultado de la instrucción?", el 55.1% de los alumnos contestó correctamente que las variables J y K son las que se almacenan en la referencia indirecta.

Pregunta 5: Para instrucciones de invocación tales como *System prints the first Number and newline*, se observó que para la pregunta "¿qué valor se imprime a partir de la instrucción?", el 51% de los alumnos contestó correctamente 25.

Pregunta 8: Dada la siguiente instrucción:

```
adjective Speed:
  attribute distance as a Real Number.
  attribute time as an Real Number.
  derived attribute result
   as a Real Number:
    distance / time; and return it.
```

Cuyo equivalente en Java es:

```
class SpeedFormula {
  float distance; float time;
  float result() {return distance/time;}}
```

El 22.4% mencionó que el lenguaje es *altamente expresivo*, el 38.8% mencionó que es *medianamente expresivo*, el 34.7% lo calificó como *poco expresivo*, mientras que el 4.1% lo describió como *nada expresivo*.

Se observa que en general, el lenguaje obtuvo un buen nivel de aceptación entre los encuestados, quienes al final mencionaron que la mayor limitante es el idioma, algo entendible si se considera que tan sólo un encuestado mencionó que el inglés es su lengua materna. Otro problema que se observó radica en que los encuestados, al tener conocimientos de lenguajes como Java, tienden a comparar y a estructurar las ideas en el paradigma orientado a objetos, lo que da como resultado una resistencia al cambio que se previó, ya que es un fenómeno que ocurre al surgimiento de un nuevo paradigma. La mayor prueba de esta resistencia se observó cuando se trató de explicar la encapsulación de código disperso entre objetos con AspectJ para su comparación con las circunstancias de SN, ya que se observó que los encuestados no pudieron asimilar el concepto con facilidad ya que en su experiencia, el código se entiende mejor con la dispersión y no se requiere encapsular.

VII. Conclusiones y Trabajo a Futuro

En este artículo se presenta un modelo de programación naturalística de propósito general que se obtuvo a partir de un análisis tanto de diversos trabajos, como de la propia estructura de la lengua inglesa. Dicho modelo se utilizó para diseñar el lenguaje naturalístico SN, que posee soporte para referencias indirectas de tipo temporal, estructural y reflexiva. Además, SN permite un mecanismo de composición en tiempo de instanciación por medio de la combinación de sustantivos con adjetivos; permite el uso de circunstancias para definir eventos; por último, posee un nivel de expresividad que se basa en el uso de sintagmas. SN se diseñó para comprobar si el modelo naturalístico contribuye a reducir la brecha entre el dominio del problema y el dominio de la solución considerando la parte formal que se requiere en un lenguaje de programación. La implementación del lenguaje SN dio como resultado un lenguaje verboso y que permite expresar instrucciones con un nivel de expresividad más cercano al idioma inglés gracias también a la gran cantidad de reglas que se utilizaron para su creación.

La progrmación naturalística basada en un modelo permite reducir la ambigüedad de los lenguajes naturales para su uso en la programación, ya que la ambigüedad de los lenguajes naturales deriva en construcciones gramaticales con una complejidad que una computadora puede resolver de forma errónea, como se presenta en [25].

Como trabajo a futuro se pretende refinar el lenguaje SN para realizar casos de prueba complejos que permitan validar de forma más completa el modelo, se pretende evaluar la expresivida de dichos ejercicios no sólo por medio de análisis subjetivo, sino también con el índice de grado Flesch-Kincaid [31] y el índice grado Gunning [32] para un análisis cuantitativo de la expresividad, aunque al ser SN un subconjunto formalizado del idioma inglés, se espera un sesgo en los resultados a partir de la complejidad gramatical de las construcciones respecto a las intenciones del programa. Como parte de los refinamientos se trabajará en la integración de un mecanismo que permita el uso de gramáticas embebidas, esto por la complejidad para describir elementos de dominios particulares de forma naturalística. Las gramáticas embebidas se consideran porque en los lenguajes naturales, los formalismos de un domino particular se encuentran separados de la redacción ya sea por el tipo de letra o inclusive por bloques donde se especifica que son formalismos y no lenguaje natural. Además, se evaluará si las instrucciones de iteración y decisión requieren de un mayor nivel de detalle en las descripciones. Se valorará la coherencia de las referencias indirectas en todas las instrucciones, dado que se observaron instrucciones tales como *repeat the first Number times...* que aunque es expresiva para describir una acción, resulta incoherente al momento de leerla. Todo lo anterior con el objetivo de comprobar que el modelo realmente sirve como base para el diseño de lenguajes formales más expresivos desde la perspectiva del programador, si se requiere de más elementos para complementarlo o si el modelo posee elementos que quizá no son necesarios.

Agradecimentos

Los autores agradecen al Consejo Nacional de Ciencia y Tecnología (CONACyT) por el apoyo otorgado para este proyecto.

Referencias

[1] J. E. Sammet, "The use of english as a programming language," *Commun. ACM*, vol. 9, no. 3, pp. 228–230, mar 1966. [Online]. Available: http://doi.acm.org/10.1145/365230.365274

[2] F. Rojas Lopez, I. Lopez Arevalo, D. Pinto, and V. J. Sosa Sosa, "Context expansion for domain-specific word sense disambiguation," *IEEE Latin America Transactions*, vol. 13, no. 3, pp. 784–789, March 2015.

[3] F. Gregory, "Cause, effect, efficiency and soft systems models," *Journal of the Operational Research Society*, vol. 44, no. 4, pp. 333–344, 1993. [Online]. Available: https://doi.org/10.1057/jors.1993.63

[4] D. Price, E. Rilofff, J. Zachary, and B. Harvey, "Naturaljava: A natural language interface for programming in java," in *Proceedings of the 5th International Conference on Intelligent User Interfaces*, ser. IUI '00. New York, NY, USA: ACM, 2000, pp. 207–211. [Online]. Available: http://doi.acm.org/10.1145/325737.325845

[5] A. W. Biermann and B. W. Ballard, "Toward natural language computation," *Comput. Linguist.*, vol. 6, no. 2, pp. 71–86, apr 1980. [Online]. Available: http://dl.acm.org/citation.cfm?id=972439.972440

[6] H. Liu and H. Lieberman, "Metafor: Visualizing stories as code," in *Proceedings of the 10th International Conference on Intelligent User Interfaces*, ser. IUI '05. New York, NY, USA: ACM, 2005, pp. 305–307. [Online]. Available: http://doi.acm.org/10.1145/1040830.1040908

[7] O. Pulido-Prieto and U. Juárez-Martínez, "A survey of naturalistic programming technologies," *ACM Comput. Surv.*, vol. 9, no. 18, pp. 17:1–17:39, apr 2017. [Online]. Available: http://doi.acm.org/10.1145/3037755

[8] R. Knöll, V. Gasiunas, and M. Mezini, "Naturalistic types," in *Proceedings of the 10th SIGPLAN Symposium on New Ideas, New Paradigms, and Reflections on Programming and Software*, ser. Onward! 2011. New York, NY, USA: ACM, 2011, pp. 33–48. [Online]. Available: http://doi.acm.org/10.1145/2048237.2048243

[9] G. Booch, R. A. Maksimchuk, M. W. Engle, B. J. Young, J. Conallen, and A. H. Kelli, *Object-Oriented Analysis and Design with Applications*, 3rd ed. Boston, MA, USA: Addison Wesley Longman, 2007.

[10] D. Janssens, "Natural language processing in requirements elicitation and requirements analysis: a systematic literature review," Master's thesis, Utretch University, Netherlands, 2019.

[11] B. W. Ballard and J. C. Lusth, "An english-language processing system that "learns" about new domains," in *Proceedings of the May 16-19, 1983, National Computer Conference*, ser. AFIPS '83. New York, NY, USA: ACM, 1983, pp. 39–46. [Online]. Available: http://doi.acm.org/10.1145/1500676.1500682

[12] M. Landhäußer, S. Weigelt, and W. F. Tichy, "Nlci: a natural language command interpreter," *Automated Software Engineering*, vol. 24, pp. 839–861, 2016. [Online]. Available: http://dx.doi.org/10.1007/s10515-016-0202-1

[13] P. Van Roy and S. Haridi, *Concepts, Techniques, and Models of Computer Programming*, 1st ed. The MIT Press, March 2004.

[14] G. Kiczales, J. Lamping, A. Mendhekar, C. Maeda, C. Lopes, J.-M. Loingtier, and J. Irwin, "Aspect-oriented programming," in *ECOOP'97 — European Conf. Object-Oriented Programming*. Berlin, Heidelberg: Springer Berlin Heidelberg, 1997, pp. 220–242.

[15] C. V. Lopes, P. Dourish, D. H. Lorenz, and K. Lieberherr, "Beyond aop: Toward naturalistic programming," *SIGPLAN Not.*, vol. 38, no. 12, pp. 34–43, dec 2003. [Online]. Available: http://doi.acm.org/10.1145/966051.966058

[16] R. Knöll and M. Mezini, "Pegasus: First steps toward a naturalistic programming language," in *Companion to the 21st ACM SIGPLAN Symposium on Object-oriented Programming Systems, Languages, and Applications*, ser. OOPSLA '06. New York, NY, USA: ACM, 2006, pp. 542–559. [Online]. Available: http://doi.acm.org/10.1145/1176617.1176628

[17] N. E. Fuchs and R. Schwitter, "Attempto controlled english (ACE)," March 1996.

[18] N. Fuchs *et al.*, *Attempto Controlled English Language Manual Version 3.0*, Geneva, Switzerland: Institut für Informatik der Universität Zürich, The address of the publisher, 1999.

[19] N. E. Fuchs, K. Kaljurand, and T. Kuhn, "Reasoning web," in *Reasoning Web*, C. Baroglio, P. A. Bonatti, J. Maluszyński, M. Marchiori, A. Polleres, and S. Schaffert, Eds. Berlin, Heidelberg: Springer-Verlag, 2008, ch. Attempto Controlled English for Knowledge Representation, pp. 104–124. [Online]. Available: http://dx.doi.org/10.1007/978-3-540-85658-0_3

[20] T. Kuhn and A. Bergel, "Verifiable source code documentation in controlled natural language," *Sci. Comput. Program.*, vol. 96, no. P1, pp. 121–140, dec 2014. [Online]. Available: http://dx.doi.org/10.1016/j.scico.2014.01.002

[21] P. Clark, W. R. Murray, P. Harrison, and J. Thompson, "Naturalness vs. predictability: A key debate in controlled languages," in *Proceedings of the 2009 Workshop on Controlled Natural Language*, N. E. Fuchs, Ed. Berlin, Heidelberg: Springer Berlin Hidelberg, 2010, pp. 65–81. [Online]. Available: http://dx.doi.org/10.1007/978-3-642-14418-9_5

[22] P. Clark, P. Harrison, T. Jenkins, J. Thompson, and R. Wojcik, "Acquiring and using world knowledge using a restricted subset of english," in *Proceedings of the 18th International Florida Artificial Intelligence Research Society Conference (FLAIRS 2005)*. AAAI Press, 2005, pp. 506–511.

[23] M. Mefteh, N. Bouassida, and H. Ben-Abdallah, "Towards naturalistic programming: mapping language-independent requirements to constrained language specifications," *Science of Computer Programming*, vol. 166, pp. 89–119, 2018.

[24] A. S. Bruckman, "Moose crossing: Construction, community, and learning in a networked virtual world for kids," Ph.D. dissertation, Cambridge, MA, USA, 1997.

[25] A. Bruckman and E. Edwards, "Should we leverage natural-language knowledge? an analysis of user errors in a natural-language-style programming language," in *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, ser. CHI '99. New York, NY, USA: ACM, 1999, pp. 207–214. [Online]. Available: http://doi.acm.org/10.1145/302979.303040

[26] A. Cozzie, M. Finnicum, and S. T. King, "Macho: Programming with man pages," in *Proceedings of the 13th USENIX Conference on Hot Topics in Operating Systems*, ser. HotOS'13. Berkeley, CA, USA: USENIX Association, 2011, pp. 7–7. [Online]. Available: http://dl.acm.org/citation.cfm?id=1991596.1991606

[27] J. Keim and A. Koziolek, "Towards consistency checking between software architecture and informal documentation," in *2019 IEEE International Conference on Software Architecture Companion (ICSA-C)*. IEEE, 2019, pp. 250–253.

[28] J. Keim, Y. Schneider, and A. Koziolek, "Towards consistency analysis between formal and informal software architecture artefacts," in *Proceedings of the 2nd International Workshop on Establishing a Community-Wide Infrastructure for Architecture-Based Software Engineering*. IEEE Press, 2019, pp. 6–12.

[29] R. Likert, "A technique for the measurement of attitudes." *Archives of psychology*, vol. 140, pp. 1–55, 1932.

[30] N. Chomsky, "Three models for the description of language," *IRE Transactions on information theory*, vol. 2, no. 3, pp. 113–124, 1956.

[31] J. P. Kincaid, R. P. Fishburne Jr, R. L. Rogers, and B. S. Chissom, "Derivation of new readability formulas (automated readability index, fog count and flesch reading ease formula) for navy enlisted personnel," Institute for Simulation and Training, University of Central Florida, Tech. Rep., 1975.

[32] R. Gunning, "The fog index after twenty years," *Journal of Business Communication*, vol. 6, no. 2, pp. 3–13, 1969.

Oscar Pulido Prieto tiene el grado de Ingeniero en Sistemas Computacionales (2011), Maestro en Sistemas Computacionales (2014), Doctor en Ciencias de la Ingeniería (2019) por parte del Tecnológico Nacional de México – Instituto Tecnológico de Orizaba. Tiene experiencia en ingeniería de software, diseño de herramientas de modelado para ingeniería orientada a aspectos y programación naturalística.

Ulises Juárez Martínez hizo su investigación doctoral sobre la programación orientada a aspectos con capacidades de corte en variables locales, obteniendo una plataforma para la verificación de aseveraciones a tiempo de ejecución. Actualmente es profesor investigador de la División de Estudios de Posgrado e Investigación del Tecnológico Nacional de México Instituto Tecnológico de Orizaba. Ha dirigido proyectos de investigación, tesis de nivel maestría y doctorado aplicando la orientación a aspectos en ingeniería de software, líneas de productos de software y en la implementación de lenguajes naturalísticos de propósito general.