Programming Spatial Algorithms in Natural Language

Boris Galitsky[1] **and Daniel Usikov**[2]

[1]Knowledge-Trail Inc. 9 Charles Str. Natick MA 01760
[2]Electroglas, Inc. 5729 Fontanoso Way San Jose, CA 95138

Abstract

We attempt programming spatial algorithms in natural language. The input of the proposed system is a natural language description of a spatial processing algorithm, and the output is the object-oriented program code to be compiled and executed.

Two approaches are proposed and evaluated: the first one is based on textual pattern matching: the best fit pattern is selected for each sentence, and objects and methods are instantiated according to this textual pattern. The second one converts text into logic forms subject to a number of transformations to derive resultant code. A number of heuristic rules are built to perform these transformations at every step. Additional pass is then required to process sentences' coreferences to find identical objects, methods and variables between code statements.

The result of this preliminary research suggest that the using the system for programming in natural language in the interactive mode, where a user manually edits the generated code, is a way to noticeably increase coding performance. However, at this time, an accuracy of a fully automated (i.e. non-interactive) code generation mode is still too low to be usable.

Introduction

More than 30 years ago Edsger W. Dijkstra, a Dutch computer scientist who invented the concept of "structured programming", wrote: "I suspect that machines to be programmed in our native tongues -be it Dutch, English, American, French, German, or Swahili-- are as damned difficult to make as they would be to use" (Dijkstra 1986). The visionary was definitely right – the specialization and the high accuracy of programming languages are what made possible the tremendous progress in the computing and computers as well. Dijkstra compares the invention of programming languages with invention of mathematical symbolism. In his words "Instead of regarding the obligation to use formal symbols as a burden, we should regard the convenience of using them as a privilege: thanks to them, school children can learn to do what in earlier days only genius could achieve". But thirty years later we again hit a wall – this time with the amount of code sitting in a typical industry applications – tens and hundreds of millions lines of code, - a nightmare to support and develop. The examples are abound, with Microsoft as an immediate call with their burden of legacy and countless postpones of new releases. The classical: "The code itself is the best description" became kind of a bad joke. May be there is a good time to ask again: how about natural language programming?

Recently, the researchers from MIT Media Lab came with a project "Metafor", - an attempt to translate (simplified) sentences of simple scenarios in natural language directly into one of the high-level object programming languages (Liu & Lieberman 2005). However, rather complex means are required to understand the program, which contains interconnected set of expressions which are queries, declarations, definitions and imperative statements.

In this study we introduce an interactive code development environment for programming in natural language (PNL) and perform its initial evaluation. We select the domain of spatial knowledge processing and image understanding algorithms because it is rather narrow and allows compact formalizations. It has been noticed that there is a good match between the structure of natural language and object-method relations in object-oriented programming (Pane and Myers 2004). As a result, the image objects could be relatively easily translated into objects of an object-oriented programming. In general, programming in natural language (without any constrain on the syntax) could seem impossible to implement because it would require full understanding of natural language and dealing with the peculiarities of human natural language descriptions. However, we believe in a properly circumscribed domain, reasoning can compensate for natural language ambiguity, since the number of entities and possible meaning for each of these entities is limited in such domain as spatial knowledge processing. Primarily, reasoning about coreferences (reference in one expression to the same referent in another expression) is important to combine code representation of individual NL statements.

The advantage of PNL in comparison with the conventional (formal language) programming is primarily ease of code sharing: it is much easier to understand someone else's code written in the natural language. Hence a teamwork would benefit from programming in natural language, and, overall, it would become possible to design much more complex systems (particularly, AI systems). Universality, brevity, intuitiveness and inherent possibility

Copyright © 2008, Association for the Advancement of Artificial Intelligence (www.aaai.org). All rights reserved.

of high-level programming are among the other features of the desired programming in natural language.

In this article we make an attempt to map a natural language description into the sequence of method invocation for a set of software objects. We follow this methodology in general, but augment it with representation of input text by means of logic forms to handle. It allows controlling the above mapping directly into the program, separate natural language expressions for special logical constructions of programming language. These constructions include branching and loops, signatures of functions (which variable is known, which is checked and which value is to be obtained), and other features of programming languages existing independently of object-orientation properties.

The paper is organized as follows. We start with simple examples of algorithms in NL such as factorial and demonstrate direct mapping from parts of speech to control operators, objects, and their methods and attributes. A pattern matching algorithm is then presented which facilitates more complex transformation from NL to the programming code. A spatial reasoning domain is then introduced, and a logic form (rule-based) approach is employed. Both approaches are evaluated by means of building a series of edge detection algorithms, and the conclusion is drawn about satisfactory code accuracy if used within an integrated development environment.

Mapping natural language statements into the code

We first demonstrate that for simple algorithms expressed in NL we can translate from NL to Java directly. The purpose of this section is to show an easy part of NL program understanding: how parts of speech are mapped into control operators and objects with their methods. We start with a basic definition of a factorial function (via loop and recursive, traditional for introduction to programming) Figure 1a and 1b.

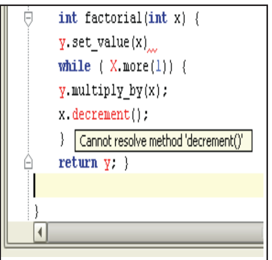
NL	Java
We are Defining new method *factorial*	
It gets an integer X and returns integer Y.	int factorial(int x) {
Set a value of Y to X.	y.set value(x)
While X is more than 1	while (X.more(1)) {
Multiply Y by X	y.multiply_by(x);
Decrement X	x.decrement();
End While	}
Return Y	return y; }

We now proceed to recursive definition

NL	Java
We are Defining new method *factorial*	
It gets an integer X and returns integer Y.	int factorial(int x) {
Check if X is less or equal than 1	if x.less_or_equal(1)
Return 1	return 1;
Else return multiply X by factorial of the decrement of X	else return x.multiply_by(factorial(x.decrement));
Decrement X	x.decrement();
End While	}
Return Y	return y; }

Figure1a. Code for Factorial function.

The words to code mapping required to build the above algorithms follow the context-free grammar and are as follows (Fig.1b). The Java IDE shown on the right, requesting to define *decrement* method.

	Java	
We are Defining new method *factorial*		
Gets integer X	…(int x) {	
Check if X	if x…	
Return 1	return 1;	
multiply X by factorial of	x.multiply_by (factorial(…));	
While X	while (X …) {	

```
int factorial(int x) {
y.set_value(x)
while ( X.more(1)) {
y.multiply_by(x);
x.decrement();
}
return y; }

}
```
Cannot resolve method 'decrement()'

Figure 1b. Required mappings and IDE screen-shot

There are two passes: the first one just builds object, methods and attributes for every part of speech in the sentence without worrying whether it is known to the system, and then the second pass identifies unknown variables and requests to provide definition for them. In this case second pass does not modify a program.

Domain of image analysis

PML required building an ontology including four components:
1. It is assumed, that the entities for basic arithmetic and logic operations are already incorporated in the programming platform;
2. It is also assumed, that the entities for object-oriented programming terms (which are words in natural language), like "objects", "methods", "attributes", "declarations", etc) are already incorporated in the programming platform;
3. Indefinable task-oriented/domain dependent entities, which are not defined via the entities of the components 1, and 2 above. These can be seen as extended types of variables in the standard programming languages which are handled by additional software libraries.

4. Derived entities, i.e. task oriented objects (in our example - the spatial domain) which are defined (or derived from) from the entities of above components 1,2, and 3.

Naturally we start defining spatial objects with the indefinable entities, such as **pixel**, allowing the construction of any image. Pixel has attributes such as its position in the image (X,Y coordinates) and brightness. Pixel is an example of indefinable domain-specific entity, which belongs to the above defined domain 3.

Below we show examples of *derived* entities, i.e. belonging to the component 3. We will then specifically show how an image algorithm expressed in the natural language (using all components 1,2,3,4) could be transformed into a code written in one of the standard object-oriented languages (Java).

We proceed to the examples defining the derived entities, with the binary images being used in the examples as follows.

On-pixel – pixel with brightness 128 or greater.

Off-pixel – pixel with brightness less than 128.

Image – a set of pixels to work with.

Area – a set of pixels (usually all having a specific property -- for instance, consisting of off-pixels).

Neighboring pixels for p – four-tuple of pixels p1, p2, p3, p4 which are above, below, on the right, or on the left of p. The "four-neighbors" definition is assumed as a default for the word (i.e. object) neighbor, but it can be an "eight neighbors" as well, if the attribute has been explicitly changed.

Connected area – a set of pixels so that each two pixels p1,p2 of the area, are connected via a chain of neighbors.

Line segment connecting pixels p1 and p2 (let's call it "p1-p2") is a minimal connected area (i.e. having the minimal number of pixels) including the pixels p1 and p2. The length of the line segment is equal to the number of pixels in the line segment (default). The line segment length attribute might be changed to the Euclidian distance between pixels p1 and p2.

Pixel p is located **between** pixels p1 and p2 if it belongs to the p1-p2 line segment and does not coincide with p1 or p2.

Convex area – a set of pixels with a property that for any two pixels p1 and p2 from the set, any pixel located on the line segment p1-p2 belongs to the set. The line segment is a convex area itself.

Edge of a connected area is a set of pixels which among their neighbors have at least one pixel not belonging to the set. In other words, the edge consists of the "outmost" pixels of the area.

Correspondingly, **border** of a connected area is a set of pixels which do not belong to the area, but among their neighbors have at least one pixel which does belong to the area.

Size of an area is a maximum distance between two pixels of the area.

Above and other definitions are covered by five syntactic templates, which are the intermediate step before building logical forms (Fig. 2, Galitsky 2003).

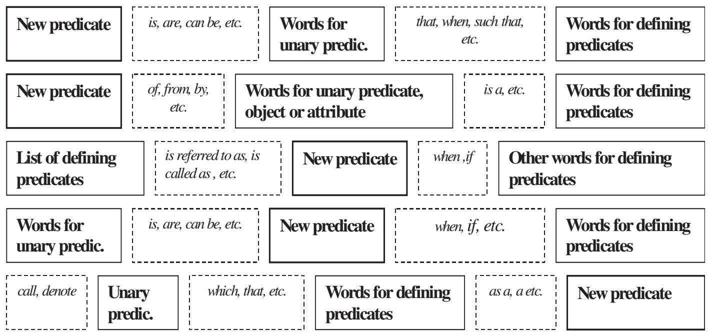

New predicate
is, are, can be, etc.
Words for unary predic.
that, when, such that, etc.
Words for defining predicates
New predicate
of, from, by, etc.
Words for unary predicate, object or attribute
is a, etc.
Words for defining predicates
List of defining predicates
is referred to as, is called as , etc.
New predicate
when ,if
Other words for defining predicates
Words for unary predic.
is, are, can be, etc.
New predicate
when, if, etc.
Words for defining predicates
call, denote
Unary predic.
which, that, etc.
Words for defining predicates
as a, a etc.
New predicate

Figure 2. Selected syntactic templates of generic definitions, formed for their recognition. Lexical units are schematically depicted which will be interpreted as predicated: the new one, being defined, the unary one, naming the introduced object, and the defining ones. Examples of linking words are presented.

Having defined the entities, PNL now allows introduction of an example *algorithm* for object finding in an image. As an example, we choose the following algorithm for finding a connected off-pixels area with a constrained size:

1) Find in the image any pixel p1 of type "off-pixel". Call the pixel a_off area.
2) For all border pixels of a_off area find all pixels of "off-pixel" type. Add them to a_off area.
3) Verify that the border of the a_off area has all pixels above 128 (i.e. consisting of pixels of "on-pixel" type).
4) If the above verification succeeds, stop with positive result. Otherwise, add all pixels which are below 128 to the *a_off*.
5) Check that the size of *a_off* is below the threshold. Then go to 2. Otherwise, stop with negative result.

Using placeholders (such as *a_off*) for objects and attributes, explicit variables, etc. instead of pronouns decreases the complexity of syntactic analysis of a definition, as well as the formation of the resultant clause. For example, if in definitions above we use p1 and p2 for the denotation of pixels that would make the definition structure clearer and easier to understand for both human and com

Pattern matching algorithm

In this section we will observe how an algorithm description can be converted into code using surface-level understanding of text using pattern matching. We assume that a natural language definition consists of two components of a (complex) sentence, one of which contains the predicate (word combination) being defined, and the second component begins from *if*, *when*, *as only as* and includes the defining predicates.

To map a NL statement into a code statement, we use a *duple*, which can be matched against a sentence and form a generic code expression. A duple includes

1) *NL matching* component, which verifies if a NL statement contains indications of control operators, objects, their methods etc., and
2) G*eneric code structure* to instantiate execution control operators, objects, method invocation, definition, as well as an execution control.

The components in the duple are connected so that the names of classes, variables, and methods, are instantiated correspondingly.

NL matching component is expected to be a unique match for every valid PNL expression. It is specified as an encoded parsing tree, or as a sequence of keywords, where some keyword placeholders specify constraints on part of speech or other linguistic parameters.

As a simple example of duple, the template {_verb, _variable1, "by", variable2} is mapped into a method (usually described as a verb) *_verb* which is applied to *_variable1* with parameter *_variable2*. Here *_term* denotes a placeholder for a keyword with certain constraints (such as part-of-speech, enumeration, and others). The code for this expression is *_variable1._verb(_variable2).* Let us look at the generic template and its instantiation for the above case

The NL template component of the simple duple is on the top, and the code component is on the bottom. Direct mapping (instantiation) is shown by arrows.

Let us now introduce a generic duple which covers all basic programming expression. The NL template component, starting with *control wrapper* for execution control (returning, jump, or possibly a start of declaration), is shown on the top, followed by the type of variable, the variable itself, the object, the method, and a list of its arguments (Fig. 3-1, 3-2).

We now proceed to the examples outlined above in our definition of Factorial. The above NL matching template would match the expression "Multiply Y by X" (Fig. 3-3).

We are Defining new public method Factorial. It gets an integer X and returns integer Y.

{"define", _declaration-directives, _method-name, "get", _type1, _variable1, "return", _type2, _variable2}

For X =Y1:Y2 {"for", _variable, "equal", "to", _variable1, "to", _variable2} - Fig. 3-4).

Here we need to use a generic method *loop* which implements increments of Y1 till it reaches Y2.

For X =Y1:Y2 such that X is satisfied (i.e. is valid)

{"for", _variable, "equal", "to", _variable1, "to", _variable2, "such", "that", _variable, "satisfies". _function } – Fig. 3-5).

Hence the syntax of *NL template* expressions is as follows. It is a sequence of terms, each of which is:

- a keyword, which can be noun, verb, adjectives, adverbs, and prepositions;
- a placeholder for a NL terms with certain linguistic constraint; these are used to instantiate the PL structure;
- a placeholder for an arbitrary words, which can be substituted by any parts of speech or not substituted at all.

Since our templates are syntactic, the complexity of expressions it can handle is limited. In the section to follow we introduce semantic means to deal with more complex expressions to be converted into code.

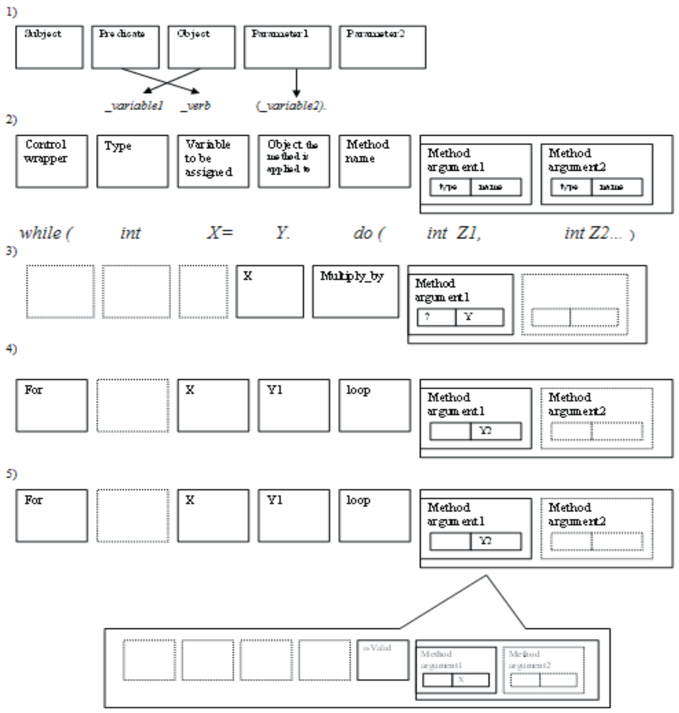


Figure 3. Duples for pattern matching.

Building the code fragment via logic forms

Textual pattern matching can perform satisfactorily in a relatively simple code with rather short NL statements. To approach a realistic software development environment, we build a number of intermediate representation layers (Galitsky 2003) to apply a series of semantic transformation rules, enumerated below at a high level.

Using logic forms for semantic analysis, in addition to the set of indefinable entities, the domain ontology should be built, including:

a) Entities, represented as logic predicates (for indefinable entities);
b) Their parameters, represented as arguments of these predicates;
c) Clauses for available set of definitions.

We illustrate a step-by-step conversion of a natural language expression into an object-oriented program. At a high level, there are two major analysis levels for this conversion:

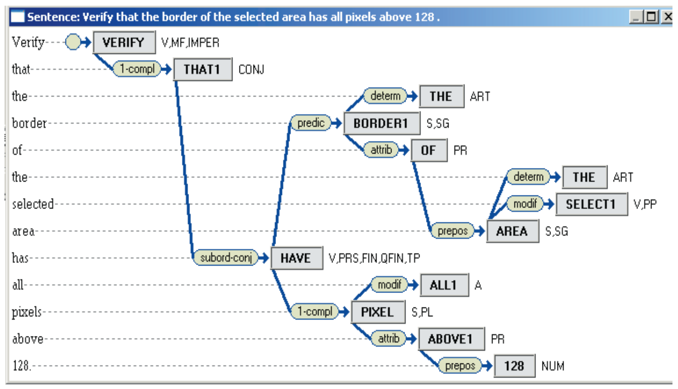
1) Building a parsing tree of each sentence;
2) Building logic forms from the parsing tree;
3) Forming expressions for objects' methods, given (1) and (2).

At a more specific level, there are the following steps:

1-1) Identify syntactic relationship between words, including relationships:
   1-1-1 Predicate-argument
   1-1-2 Predicate-predicate
   1-1-3 Epistemic action (request to find or verify a value or logical condition).

1-2) Represent syntactic tree as a list of pairs of linked words.

2-1) Rewriting the natural language sentence grouping words by logic predicates. We express predicates with arguments as a sequence of NL expressions (tuples of words for predicates and their arguments);
2-2) Reorganize words from the tuples into logical predicates and their arguments;
2-3) Converting all constants into variables, we attempt to minimize the number of free variables, and not over-constrain the expression at the same time;
2-4) Adding predicates which constrain free variables;
2-5) Quantification;
3-1) Mapping predicates and their arguments into the objects, their methods and arguments:
3-2) Finding code template for specified epistemic action
3-3) Building the resultant code fragment


Sentence: Verify that the border of the selected area has all pixels above 128 .

Word	Link	Node	Tags
Verify		VERIFY	V,MF,IMPER
that	1-compl	THAT1	CONJ
the	determ	THE	ART
border	predic	BORDER1	S,SG
of	attrib	OF	PR
the	determ	THE	ART
selected	modif	SELECT1	V,PP
area	prepos	AREA	S,SG
has	subord-conj	HAVE	V,PRS,FIN,QFIN,TP
all	modif	ALL1	A
pixels	1-compl	PIXEL	S,PL
above	attrib	ABOVE1	PR
128.	prepos	128	NUM

Figure 4. Each word is assigned a graph node, parent node is specified, and recognized part-of-speech parameters are indicated. This serves as a basis for deciding which word is mapped into a predicate, its argument, or neither. We group parts of speech to be mapped into the predicates, using such expressions as verb-object, subject-has-object, object-related to-value, and others.

We will now show this process for a single sentence from the above algorithm description. Given the sentence "Verify that the border of the selected area has all pixels above 128", we yield its syntactic structure (Fig. 4).

Below the arrows show how the sequence of mapping is defined to map original NL expression into a logic form, and then into a fragment of code.

```
Verify that the border of the selected area
has all pixels above 128.
```

2-1) We express in predicate with arguments as sequences of NL expressions

```
Verify:Verb + border -of- area +  border –
have- pixel + pixel above 128
```

2-2) And now code it as predicates

```
epistemic_action(verify) & border(area) &
border(pixel)& above(pixel, 128)
```

2-3) Convert constants into variables with proper generality control

```
epistemic_action(verify) & border(Area) &
border(Pixel) & above(Pixel, 128)
```

Converting all constants into variables, we attempt to minimize the number of free variables, and not over-constrain the expression at the same time. Coupled (linked by the edge) arrows show that the same constant values (pixel) are mapped into equal variables (Pixel), following the conventions of logic programming. To achieve this, we add (unary) predicates which need to constrain free variables.

2-4) Adding predicates which constrain free variables

```
epistemic_action(verify) & border(Area) &
border(Pixel) & above(Pixel, 128) &
area(Area)
```

Now we need to build an explicit expression for quantification **all.** In this particular case it will not be in use, since we use a loop structure anyway

2-5) Quantification

```
epistemic_action(verify) & border(Area) &
not ( border(Pixel) & not above(Pixel, 128))
& area(Area)
```

3-1) Next step is to map predicates and their arguments into the objects, their methods and arguments:

```
Loop => Pixel.next()
border.belong(Pixel) && Pixel.above(128)){
```

Finally, the expression can be transformed into a loop, since epistemic_action is 'verify'. We have the following template for it.

3-2) Finding code template for specified epistemic action

```
Bool bOn=true;
while (!(ObjectToTest.next()==null)) {
if !(Conditions){
  bOn=false;
      break;}
} Return bOn;
```

Finally, we have

3-3) Resultant code fragment

```
while (!(Pixel.next()==null)) {
if !(border.belong(Pixel) && Pix-
el.above(128)){
  bOn=false;
      break;
       }
}
```

```
Return bOn;
```

Notice that in a general case, not all mappings applied to all entities and predicates might be required, and some mappings are redundant.

Constructing the code for the whole algorithm

Determining if a larger area is an off –area (lower brightness). We start with a randomly selected pixel

0) Randomly select a pixel at an image.

`epistemic_action(random_selection), pixel(PixelSelect).`

1) Find a convex area a_off this pixel belongs so that all pixels are less then 128.

`epistemic_action(find) & convex_area(A_off)` **`not (`** `belong(Pixel, A_off) &` **`not`** `above(Pixel, 128))`

2) Verify that the border of the selected area has all pixels brighter than 128.

`epistemic_action(verify) & border(Area) &` **`not (`** `border(Pixel) &` **`not`** `above(Pixel, 128))` `& area(Area)`

3) If the above verification succeeds, stop with positive result. Otherwise, add all pixels which are below 128 to the a_off.

`convex_area(A_off)`(obtained from the above) `ifthenelse (epistemic_action(succeeds-verify), return true,` **`not (`** `border(Pixel) & not above(Pixel, 128), add( )`

4) Check that the size of a_off is below the threshold. Then go to 2) above. Otherwise, stop with negative result.

`epistemic_action(verify) & ifthenelse( (size(Area, Size), less(Size, thre)), goto(2), return false;`

Actually, we need to the statement 2) and insert `while…do` to avoid jumps.

We now proceed to the final code for each statement

```
0) epistemic_action(random_selection),
   pixel(PixelSelect).
   Pixel pixel = new Pixel(rand)

1)epistemic_action(find) &
    convex_area(A_off) not ( belong(Pixel,
    A_off) & not above(Pixel, 128))
  Area A_off=new Area(Pixel);
    while (NewPixel=Pixel.next().
       intensity()>128) {
          A_off.add(NewPixel); }
```

Everything which appears new implicitly requires a constructor. An appropriate constructor is selected based on objects available after previous statements

```
   3) convex_area(A_off)
ifthenelse (epistemic_action(succeeds-
verify), return true,
not ( border(Pixel) & not above(Pixel, 128),
add(A_off, Pixel )
Border border= new Border(A_off);
while(BorderPixel = border.next()
.intensity()>128
border.add(BorderPixel)

   3) epistemic_action(verify) & border(Area)
& not ( border(Pixel) & not above(Pixel,
128)) & area(Area)
if (border==A_off.getBorder() )
    return true;
else A_off.add(border).

   4) epistemic_action(verify) &
ifthenelse( (size(Area, Size), less(Size,
thre)), goto(2), return(false)
    if ( A_off.size()<thre goto 2 else re-
turn false;
```

Evaluation

For the purpose of preliminary evaluation, we selected simple image processing algorithm of edge detection. We looked at the Matlab 'edge' set of six functions for edge detection (Image Processing Toolbox).

Usually, when an algorithm is described as a web tutorial, its discourse structure is not well suited for PNL. Hence for the purpose of evaluation we selected algorithm descriptions, which contains the instructions on what needs to be done and not why it is done in a particular way. To build the corpus, we obtained NL descriptions of Matlab implementation of edge-functions from existing code comments and the code itself (Matlab 08).

Canny algorithm
The first step is to **filter out** any **noise** in the original image before locating and detect any edges.
Once a suitable **mask** has been calculated, the **Gaussian smoothing** can be performed using **standard convolution** methods.
A convolution mask **size** is selected to be much smaller than the actual image.
As a result, the mask is **slid** over the image, manipulating a **square** of pixels at a time.
To lower is the detector's sensitivity to noise, increase the width of the **Gaussian mask.**
To decrease the **localization error** in the detected edges, decrease the width Gaussian width.
The Gaussian mask used in implementation is ...
Using thresholding with hysteresis
This method uses multiple **thresholds** to **find edges**.
We begin by using the **upper threshold** to find the **start of an edge**.
Once we have a start point, we then trace the **path of the edge** through the image pixel by pixel, **marking an edge** whenever we are above the **lower threshold**.
We stop marking our edge only when the **value** falls below our **lower threshold**.

The list of entities used included: `grayscale image (I), edge, binary image with identified edge pixels as 1's and 2's otherwise (BW), derivative, gradient, filtering, direction of detection`. The functions signatures are of the form `BW = edge(I,method,thresh)`.
Each algorithm includes three high-level parts: defining what is gradient, applying gradient operation to the image, and performing the threshold operation. Below are two examples of simple description of algorithms used in PNL evaluation. Entities which formed the objects are shown in bold.

Method of edge detection	Number of statements describing algorithm step	Number of errors indicated by compiler and forced to be fixed		Number of remaining errors, unidentified by compiler	
		PM	LF	PM	LF
Sobel	6	8	6	2	2
Prewitt	5	5	3	2	2
Roberts	11	7	4	3	1
Laplacian of Gaussian	6	4	1	2	1
Zero-Cross	8	5	3	1	1
Canny	5	4	4	2	0

Table 1: Quality of the edge detection code generated from textual description in the interactive mode.

Results of preliminary evaluation are shown in Table 1. The reader observes that LF always outperforms PM even in the simple cases. Also, significant number of misunderstandings are identified by the Java IDE. Regretfully, even for simple algorithms the built code contains an error or two, which would produce either runtime error or produces incorrect data. They should be corrected manually. However, we believe it is a substantial aid to developers even at the current code generation accuracy because the code is "ideally" commented; there is a good match between textual description statements and lines of code.

Related work and conclusions

Natural language understanding of texts about space and required reasoning has been addressed in a number of studies. One of the closest area to PNL is querying spatial databases (see e.g. Frank 1992., *COSIT93,* Egenhofer & Shariff 1998). Widely used in spatial algorithms, primary linguistic markers of space are spatial prepositions (*in, on, under, below, behind, above*) and verbs of motion (*arrive, cross*) were shown (Herkovits 1986) to be ambiguous as purely geometric terms, and that their interpretation depends heavily on contextual factors. General theories of spatial reference and interpretations of prepositions were developed in (Clementini & Di Felice 1998), where spatial information also involves specifying spatial location from the perspective of a speaker or hearer in the communicative context, functional elements such as the typical functions of objects, and the physical nature of objects. Obviously, it is rather hard to include such forms of spatial reference in our ontology for code building, however as long as the NL description of algorithm defines all necessary attributes of involved objects, an adequate code would be generated.

In our further studies we plan to consider a wider set of spatial algorithms and a higher diversity of human descriptions. (Mainwaring et al 2003) studies how people describe the location of a target object relative to other objects. This task requires a reference object or frame and terms of reference. Traditional linguistic analyses have loosely organized perspectives around people, objects, or environments as reference objects, using
reference terms based on a viewpoint or the intrinsic sides of an object, such as left, right, front,
and back or based on the environment, such as north, south, east, and west. In actual communication, social, spatial, and cognitive factors may also affect perspective choice; these factors were examined by varying the spatial information. We will observe which of these factor can be reflected in generated code.

In this work we explored the opportunity to employ text understanding to increase the efficiency of software development. We observed that from the standpoint of general text understanding problem, building representation for explicit enumeration of tasks to be performed is not as hard as an arbitrary text understanding. The assumption that enumeration of tasks has an object-method-attribute structure allowed a simple pattern matching approach to produced meaningful results for simple algorithms. Furthermore, a rule-based semantic analysis, applied at the level of individual statements and whole description, delivered chunks of code which were improved using conventional programming means such as Java IDE. Hence we believe the PNL system described here is a reasonable step towards a fully automated code generation system.

References

COSIT93 Proceedings of the European Conference on Spatial Information Theory *(COSIT'93)*, volume 716 of *Lecture Notes in Computer Science*. Springer-Verlag, 1993.
Clementini, E. & Di Felice P Topological Invariants for Lines IEEE Transactions on Knowledge and Data Engineering archive Volume 10 , Issue 1, 1998.
Dijkstra, E.W. On the Foolishness of "Natural Language Programming". Program Construction, 51-53, 1978.
Herkovits, A. *Language and Cognition*. Cambridge University Press, New York, 1986.

Liu, H. & Lieberman, H. Metaphor: Visualizing Stories As Code. Intelligent User Interfaces Conference (IUI 2005), San Diego CA 2005. web.media.mit.edu/~hugo/publications/papers/IUI2005-metafor.pdf;

Pane, JF & Myers, BA More Natural Programming Languages and Environments, in *End User Development*, vol. 9 of the Human-Computer Interaction Series, In Lieberman, H., Paterno, F. and Wulf V., eds. Dordrecht, The Netherlands: Springer, 2006, pp. 31-50.

Cole, R., Mariani, J., Uszkoriet H., Zaenen, A. and Zue, V. (Eds) Representations of Space and Time. In *A Survey of the State of the Art in Human Language Technology. Cambridge University Press*, 1996.

Egenhofer MJ, Shariff, RBM Metric details for natural-language spatial relations. ACM Transactions on Information Systems. Volume 16, Issue 4 295 – 321, 1998.

Frank, A. 1992. Qualitative Spatial Reasoning about Distances and Directions in Geographic Space. *Journal of Visual Languages and Computing* 3, 4, 343-371.

Galitsky, B. Natural Language Question Answering System: Technique of Semantic Headers. Advanced Knowledge International, Australia 2003.

Matlab 08 www.mathworks.com Last downloaded Apr 2, 2008.

Mainwaring, SD, Tversky,B., Ohgishi, M., Schiano, D.J. Descriptions of Simple Spatial Scenes in English and Japanese. *Spatial Cognition & Computation,* Volume 3, Issue 1 March 2003, pages 3 – 42.