Fast and Reliable Software Translation of Programming Languages to Natural Language

Iaakov Exman and Olesya Shapira
Software Engineering Dept., The Jerusalem College of Engineering – JCE - Azrieli, POB 3566, Jerusalem, Israel

Keywords: Software Translation, Source Code, Natural Language, Programming Languages, Software Tool, Fast Translation, Generality, Relevance and Reliability.

Abstract: An experienced software professional with several years of programming in some languages is usually expected to read or write with proficiency in a new programming language. However, if severe time constraints are involved, and given the current availability of internet sources, there is no reason to avoid shortcuts supporting fast translation of source code keywords into Natural Language. This work describes our tool coined PL-to-NL Translator, the main ideas behind it, and its extensions. One basic assumption that was clear from the beginning of this work is the need to keep as far as possible a clear separation between generic infra-structure and the specifics of particular programming languages. Moreover, the tool keeps its generality relative to programming languages, enabling through its contributor engine, addition of any desired current or future programming language. The ideas and the software tool characteristics are illustrated by some case studies involving a few sufficiently different programming languages.

1 INTRODUCTION

There a few different motivations to translate source code in a given programming language to natural language, as an explanation of a poorly understood code:

- *Experienced programmer with new language* – a programmer with several years of programming in some languages, needs to read or write a program in a new language;
- *Novice learning* – a student or a newly hired recent graduate, needs to learn his first or second programming language;
- *Legacy program lacking documentation* – say someone needs to substitute a previous worker that moved to another company.

In all the above cases time may be a critical resource and one cannot learn the new/old programming language in a linear leisurely way. One needs to find shortcuts to comprehend in a short time, not the whole language, but the specific limited task one was assigned.

This work describes a software tool PL-to-NL (standing for "Programming Language to Natural Language) Translator which is able to translate fast and reliably code fragments in a programming language, say C++, to plain explanations in a natural language, say English.

Software professionals are familiar with programming languages as formal means of expressing programs to be run by computers. Programming languages are strictly restricted by their syntax.

Program documentation and any explanations of usage of programming languages are expected and given in natural language. This is one of the many roles of natural language related to software, which is the topic of the next paragraphs.

1.1 The Roles of Natural Language Concerning Software

We are not looking at this issue in its most general sense, as space limitations prevent us to deal with it. It suffices here to list some possible roles, and point out the relevant role in this work.

There are at least three possibilities regarding this issue, concisely formulated as follows:

1. *Highest level of software abstraction* – we have claimed elsewhere (Exman and Plebe, 2015), (Exman and Iskusnov, 2014, 2015) that natural language concepts are the highest

Exman, I. and Shapira, O.
Fast and Reliable Software Translation of Programming Languages to Natural Language.
DOI: 10.5220/0006081200570064
In *Proceedings of the 7th International Workshop on Software Knowledge (SKY 2016)*, pages 57-64
ISBN: 978-989-758-202-8

level of software abstraction, viz. one can obtain UML class diagrams from application ontologies;
2. *Software Assets Explicitly in Natural Language* – there are software assets, besides programs, say Software Requirements, that explicitly start from Natural Language sentences, that may be later refined into more formal and precise formulations with a specialized restricted syntax;
3. *Documentation as Explanations* – documentation on software may exist as separate files or as inline comments within programs.

The current work focusses on the third role.

1.2 Related Work

The literature relevant to this work is naturally divided by the different motivations, mentioned in the beginning of this Introduction.

Novice programmer learning is dealt with in a few recent papers, e.g. by Corney and collaborators (Corney et al., 2014). They claim that there is a clear correlation of the ability to read and explain programs, with writing programs' capabilities. Another paper (Teague and Lister, 2014) has an interesting proposal of testing programming ability by means of reversibility: providing the students with a small code fragment and asking them to write code that undoes the effect of that code. They also conducted think aloud studies during the reversing task.

Reading other people's code is hard; there is probably a widely agreed consensus about this statement. For instance, this is the message of the personal blog by Alan Skorkin (Skorkin, 2010). But reading code is important both as a way to acquire proficiency from experienced programmers and as part of the software system development task of "code review". Hansen et al. (Hansen et al., 2013) in their paper "What Makes Code Hard to Understand?" state that programs coherent with expectations take less time to understand and the code human interpretation is closer to correct.

We now mention a few works concerning the processing of programming languages. An influential early paper – originally lectures given in 1967 – is the paper by Strachey on Fundamental Concepts in Programming Languages (Strachey, 2000). Nilson et al. (Nilsson et al., 2009) claim that data-driven parsing approaches developed for natural languages are robust and have quite high accuracy when applied to parsing of software.

An approach to documentation, different from the work in this paper, can be described under the "Living Documentation" rubric. This is exemplified by the works of (Brown, 2011) which takes executable specs to be applied also to documentation, and (Martraire, 2016) which understands living documentation as being always up-to-date.

In particular, living documentation can be implemented by means of Wikipedia, e.g. by (Krotzsch et al., 2007) with their Semantic Wikipedia and by (Yagel, 2015) which bases living documentation on Wiki with domain knowledge.

There have been efforts to build a similar interactive tool, such as (cdecl, 2016) which translates "C" code fragments to English. Another related commercial information source of potential interest is displayed in "programmers stack exchange" (stackExchange, 2016) where one finds a discussion on "how to better in explaining the code to other developers".

1.3 Paper Organization

The remaining of the paper is organized as follows. In section 2 we introduce the software architecture of the PL-to-NL Translator tool; in section 3 we describe the Relevance & Reliability ideas of the Translator; in section 4 a concise pseudo-code of the Translator algorithm is given; in section 5 case studies illustrate the Translator usage; in section 6 there is a very short description of implementation; in section 7 the paper is concluded with a discussion.

2 THE PL-TO-NL TRANSLATOR'S SOFTWARE ARCHITECTURE

Here we shortly describe the PL-to-NL Translator software architecture. First, some principles are provided, the types of users are described, the main Translator modules are presented and finally the server's class diagram is shown.

2.1 Software Architecture Principles

The central principles behind the Translator software architecture are:

1. *Long Term Information is stored in a DB* – the Translator tool is conceived as a multiple use, long term tool, which is

gradually improved by its usage and contributions from endorsed users;

2. *Generality and Flexibility in terms of Programming Languages* – we assume that programming languages will continue to be invented and combined for diverse purposes. Thus the Translator is built taking into account a clear separation of the generic infra-structure from the specifics of any given language.

2.2 Types of Users

The PL-to-NL Translator assumes two types of users:

1. ***Guest Users*** – these are the users interested in the basic purpose of the tool: to obtain translations to Natural Language of code fragments or even a single keyword from a given Programming Language;
2. ***Contributors*** – these are more experienced users, which should be endorsed by the tool administrators, and wish from time to time to contribute a new or better specific translation in a given language, or add a whole new language to the tool.

2.3 Main Translator Modules

The Translator architecture is seen in Fig. 1.

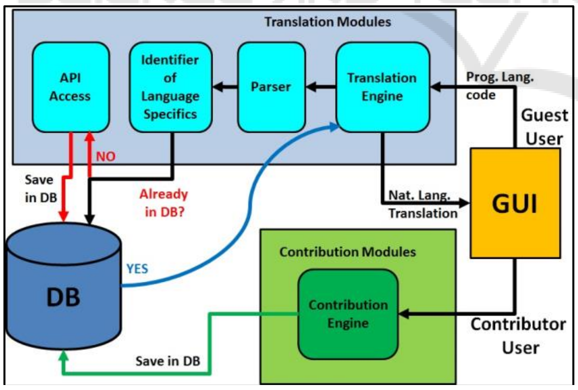


Figure 1: PL-to-NL Translator Software Architecture – Its modules are: a- GUI input/output in the right hand side; b- Translation Modules; c- Contribution Modules; d- DB, a database. Arrows point to receiver of data transmitted. The GUI is the ***client*** and all other modules constitute the ***server*** of the Translator.

The main PL-to-NL Translator modules – schematically shown in Fig. 1 – are:

1. *GUI* – the Graphical User Interface, a Web Application, is the place for input/output;
2. *Translation Modules* – whose Translation Engine receives code in a Programming Language, passes it to the Parser, and in turn to the identifier of Language Specific Features; the outcome is to check whether the translation is already in the DB;
3. *DB* – the Database stores previously known keywords etc.; these stored "translations" are sent to the Translation engine when needed and provided to the GUI;
4. *API Access* – if an identifier is not found in the DB, the relevant API is accessed and the result is saved in the DB;
5. *Contribution Engine* – if an endorsed user decides that a translation is lacking or not satisfactory, he may contribute a new translation through the contribution engine, which saves the contribution in the DB.

Further details about the classes within the modules are provided in the next subsection.

2.4 Server Class Diagram

The PL-to-NL Translator Server class diagram is seen in Fig. 2.

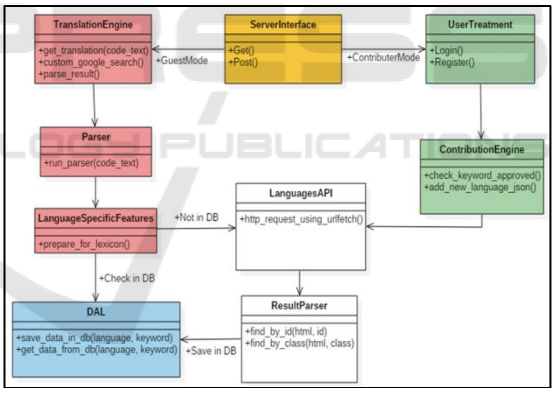


Figure 2: PL-to-NL Translator Server Class Diagram – The Server interface communicates with the Client GUI. The contribution classes are in the right-hand-side (green color). The Translation modules' classes are in the left-hand-side (pink color). The DAL class accesses the DB. In the middle one finds the API access classes.

Some interesting points about the classes in Fig. 2 are as follows:

- *check_keyword_approved* – this is a method in the Contribution Engine class; there exists for each keyword a Boolean variable, whose default value is "FALSE", until a contributor approves it, turning it into "TRUE;

- *custom_google_search* – this method in the Translation Engine class, is part of the mechanism of finding acceptable translations in the internet, as explained in the next section;
- *prepare_for_lexicon* – this method in the Language Specific Features class, prepares new added languages from "parsing DB" to the format that the parser gets as input – the lexicon.

3 THE PL-TO-NL TRANSLATOR ESSENTIAL IDEAS

Once we decide that the main source of information to translate keywords or code fragments in a given programming language to natural language is the Internet, there are two consequences:

a) ***use of commercial search engine*** – there is no reason to develop from scratch a new search engine specific for translation purposes;
b) ***efficiently filtering of information –*** the information obtained from the commercial search engine may be either useless because it is irrelevant or unreliable. Thus, efficient filtering of information quality is essential.

3.1 Main Idea: Relevance & Reliability

The main idea is to have a means to assure both Relevance and Reliability. This is the justification for the approach taken and the respective algorithm.

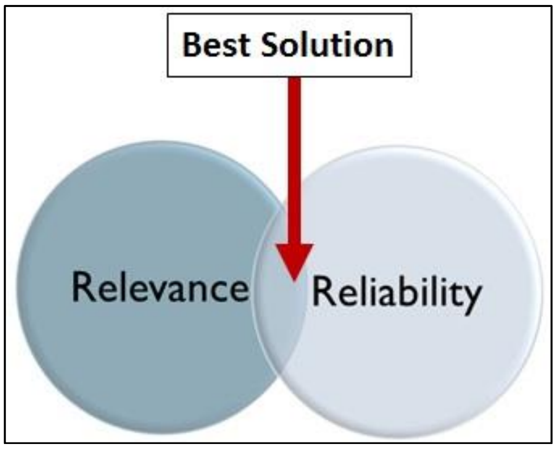


Figure 3: Relevance And Reliability – our approach to search in the Internet for sources of translation from a programming language to natural language.

Relevance implies that the web site found by the commercial search engine indeed contains the necessary information. Reliability means that one has somehow checked the quality and veracity of the information: it is true and accurate.

3.2 Approach Overview

A graphical scheme of the approach is seen in Fig. 3. It assumes that we should have a solution that combines generic search from a commercial search engine providing *relevance*, with some (partial) assurance of *reliability* from a different and independent source of chosen web-sites.

4 PL-TO-NL TRANSLATOR ALGORITHM

4.1 PL-to-NL Translator Algorithm Ideas

The Translator algorithm works in two phases:

1. ***Prepare two groups of web sites*** – one group is obtained by search for a specific language, a chosen keyword among those with a given keyword type; the other group is chosen by humans – the translator administrators and their advisors;
2. ***Loop mutually checking the two groups of web site*** – if a certain url is found in the intersection of the two groups (as in Fig. 3), this is a candidate url; otherwise choose the higher ranked url among the relevant ones.

What is the essential meaning of this algorithm?

The important point is that the best solution is the intersection of Relevance and Reliability (as shown in Fig. 3). However, if for some reason no overlap is found between Relevance and Reliability, preference is given to the relevant in detriment to the reliable ones, eventually (but not necessarily) paying a reliability price.

4.2 The Relevance & Reliability Algorithm

A pseudo-code of the Relevance & Reliability algorithm is shown in the next text-box.

The outcome RELEVANT_URLS[0] refers to the 1st member of the Relevant urls, i.e. that with the higher search rank, which is assumed to be that with the higher relevance.

```
Relevance & Reliability Algorithm

//Initialize two groups of web sites
RELEVANT_URLS  = google search by
      "<language> <keyword> <keyword type>";
RELIABLE_URLS = independent list of
approved web sites;

//Loop to obtain Relevance & Best Reliability
For ulr in RELEVANT_URLS{
     For def_url in RELIABLE_URLS{
       If (url == def_url)
           Return url;
          }}

//No Relevance & Reliability overlap
If (no url returned yet)
          Return RELEVANT_URLS[0]
```

5 CASE STUDIES

The case studies in this section illustrate the application of the Relevance & Reliability algorithm, as opposed to a simplistic approach.

5.1 Searching "Package" in Java

Suppose that we google search for:
"java package keyword site: docs.oracle.com".
The first result of the search is:

https://docs.oracle.com/javase/tutorial/java/javaOO/accesscontrol.html

which describes "Controlling access to members of a class", which has nothing to do with "package".
On the other hand, using the Relevance & Reliability algorithm, one searches for:

"java **package** keyword"

The first site from the RELIABLE_URLS that appears is:

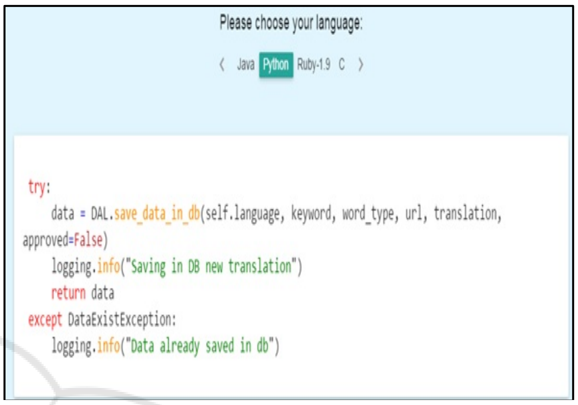
http://www.tutorialspoint.com/java/java_packages.htm

which describes exactly what we wanted.

5.2 Searching Keywords in a Python Code Fragment

Now suppose that we choose the language Python and insert a code fragment containing various types of keywords. One such example is in the partial screen-print of the PL-to-NL Translator system seen in Fig. 4. A few different types of possible keywords to be translated to natural language are:

- ***Errors and Exceptions*** – say "*try*" and "*except*";
- ***Simple Statements*** – say "*return*";
- ***Built-in functions*** – say "*info*".



```
try:
    data = DAL.save_data_in_db(self.language, keyword, word_type, url, translation,
approved=False)
    logging.info("Saving in DB new translation")
    return data
except DataExistException:
    logging.info("Data already saved in db")
```

Figure 4: Translation Of Code Fragment – One sees a PL-to-NL Translator system guest user partial screen-print, for the chosen Python language. The system marks different keyword types with different colors: language keywords such as *try* and *return* (in red), functions either built-in such as *info*, or user functions such as *save_data_in_db* (in yellow), and ignoring strings within double quotations (in green). Translation is offered only for marked language keywords and built-in functions.

This case study shows that the system is able to concurrently deal with a set of keywords in a code fragment. In this code fragment, the Translator system ignores strings (within double quotations) and function arguments. More generally, it also ignores comments.

6 IMPLEMENTATION

The PL-to-NL Translator system has been developed and actually implemented using the GAE (Google App Engine) platform.

This platform provides tools for the Server side such as a datastore, and support for the Python language. The tools provided for the client side include support for AJAX, JQuery and JS above HTML.

6.1 Parser Implementation

The parser is implemented under the following assumptions:

- Find the most common programming languages
- Find the biggest common part for most of the languages
- Build the parser based on the assumption that all languages have common features
- Find existing fundamental module; PLEX has been used, where PLEX is a Python module for constructing lexical analyzers.

The common desired features include: keywords, literals, operators, function calls and libraries.

The parser is generic. It was implemented for 3 programming languages (Java, Python, Ruby) to prove generality. The choice of these languages was dictated by their popularity among users – see e.g. (Cass, 2015).

The PL-to-NL Translator system allows the contributor to add a new language. This contributor functionality was successfully tested by the addition of the “C” language – also found in the ranking of (Cass, 2015).

7 DISCUSSION

This discussion refers to fundamental issues, comparison with other approaches and future work. It is concluded with a short statement of the main contribution.

7.1 Fundamental Issues

The following fundamental issues deserve further investigation:

a. ***Generic Programming Language Infrastructure*** – Can one define in a formal way, what is the generic common infrastructure for all the programming languages? If not, is this possible for at least certain defined families of programming languages, such as imperative or functional?

b. ***Minimal Number of Programming Languages*** – What is the minimal (or optimal) number of programming languages that a serious professional should *formally* learn? This issue is related to the previous question. One would be tempted to state that just one language would suffice, and such a professional would easily be able to learn by oneself a 2nd or a 3rd language, and so forth. But a demonstration of such a statement would involve complex cognitive functions, which are certainly beyond the scope of this paper.

c. ***Understanding Languages*** – Why is it easier to understand a freely evolving and unrestricted natural language than programming languages that have restricted syntax? This issue is raised since one usually expects program documentation and explanations to be given in natural language, despite the complexities of natural languages, such as ambiguity, metaphors, etc.

d. ***Diversity of Natural Languages*** – all this work was performed with regards the natural language “English”. How easy is to reproduce the results of the Translator for other languages? Can we find a “Generic Base” common to families of natural languages, such as Indo-European, Germanic or Slavic languages, with regard to software explanations?

e. ***Learning system*** – in principle we could insert learning capabilities into the Translator system, at least with two respects: 1- recognition of the programming language of a code fragment; 2- analysis of the natural language explanation in order to shorten it or focus it according to the perceived user interests.

7.2 Comparison with other Approaches

The main characterization of our approach to the PL-to-NL Translator is its generality and flexibility. In other words, we refer to its applicability to any programming language, provided that the chosen language has a well-defined syntax and semantics, which were published and available through the web.

The intended generality implies that we do not start with a complete and closed set of programming languages. We actually start with a small set of languages, chosen by their ranking in some popularity scale among users – e.g. (Cass, 2015). The contributor engine enables eventual adding of any future language – even not currently existing ones. This is the most stringent challenge to our approach.

The referred generality also explains why we do not just apply existing tools – e.g. parsers and their components – for existing specific languages, even if they are of the highest possible quality.

An important infra-structure feature needed for generality is to keep well-separated generic features common to various programming languages of a given family, from those features specific to a given language.

An example of an alternative approach to documentation and translation is the specific use of Wiki tools – see e.g. (Krotzsch et al., 2007), (Yagel, 2015). Despite the fact that Wiki tools have a broad enough usage, they still represent a kind of restriction to our proposed generality.

7.3 Future Work

In order to increase confidence in the PL-to-NL Translator it would be necessary to perform more extensive tests, including additional programming languages, perhaps from different families.

In principle this approach could be extended to operating systems – such as scripting languages found in UNIX or LINUX versions – and effectively to any kind of software tools with languages that may justify the investment in a Translator tool.

The system described in this paper was primarily motivated by its usefulness for an experienced programmer that has some local difficulties with a new language. It was neither intended to systematic comprehensive learning of a whole new language, nor to deal with whole long programs. This is not to say that the approach cannot be extended to these other goals. Specifically referring to whole programs, at least the GUI (Graphical User Interface) of the PL-to-NL Translator should be adapted to facilitate dealing with long inputs, instead of just small code fragments.

Finally, an investigation from the point of view of users' satisfaction should probably be performed.

7.4 Main Contribution

The main contribution of this work is the Relevance & Reliability algorithm to find in the Internet the sources to translate Programming Languages to Natural Languages. Relevance & Reliability is the basis for the generality and flexibility of our PL-to-NL Translator approach.

ACKNOWLEDGEMENT

Olesia Shapira wishes to thank Itamar Sharify for his help in originating the idea of the Programming Language to Natural Language Translator.

REFERENCES

Brown, K., 2011. "Taking executable specs to the next level: Executable Documentation", Available from: http://keithps.wordpress.com/2011/06/26/takingexecutable-specs-to-the-next-level-executabledocumentation/

Cass, S., 2015. "The 2015 Top Ten Programming Languages", IEEE Spectrum, available from site: http://www.csee.umbc.edu/courses/undergraduate/202/fall15_marron/lectures/l01/the_2015_top_ten_programming_languages.pdf

cdecl, 2016. An interactive tool to translate "C" to English. Web site: http://cdecl.org/

Corney, M., Fitzgerald, S., Hanks, B., Lister, R., McCauley, R. and Murphy, L., 2014. "'Explain in Plain English' Questions Revisited: Data Structures Problems", in SICCSE'14 Proc. of 45th ACM Technical Symposium on Computer Science Education, pp. 591-596, ACM. Web site: http://eprints.qut.edu.au/68093/. DOI: http://doi.acm.org/10.1145/2538862.2538911

Exman, I. and Iskusnov, D., 2014. "Apogee: Application Ontology Generation from Domain Ontologies", in SKY'2014, Proc. 5th International Workshop on Software Knowledge, pp. 31-42. DOI: 10.5220/0005181000310042.

Exman, I. and Iskusnov, D., 2015. "Apogee: Application Ontology Generation with Size Optimization", in Knowledge Discovery, Knowledge Engineering and Knowledge Management, Vol. 553, CCIS, pp. 477-492, Springer-Verlag, . DOI: 10.1007/978-3-319-25840-9_29.

Exman, I. and Plebe, A., 2015. "Software, Is it Poetry or Prose? Conceptual Content at the Higher Abstraction Levels", in SKY'2015 Proc. 6th International Workshop on Software Knowledge, pp. 9-17. DOI: 10.5220/0005625500050013.

Hansen, M., Goldstone, R.L. and Lumsdaine, A., 2013. "What Makes Code Hard to Understand?", Web site: http://arxiv.org/pdf/1304.5257.pdf

Krotzsch M., Vrandecic D., Volkel M., Haller H., Studer R., 2007. Semantic Wikipedia. In *Journal of Web Semantics 5/2007*, pp. 251–261. Elsevier.

Martraire C., 2016. *Living Documentation - A low-effort approach of Documentation that is always up-to-date,inspired by Domain-Driven Design*. Leanpub (expected). http://leanpub.com/livingdocumentation.

Nilsson, J., Lowe, W., Hall, J. and Nivre, J., 2009. "Parsing Formal Languages using Natural Language Parsing Techniques", in IWPT Proc. 11th Int. Conf. on

Parsing Technologies, pp. 49-60. ACM. Web site: http://www.aclweb.org/anthology/W09-38#page=65

Skorkin, A., 2010. “Why I Love Reading other People’s Code and you should too”, Personal Blog, Web site: http://www.skorks.com/2010/05/why-i-love-reading-other-peoples-code-and-you-should-too/

StackExchange, 2016. A discussion about: “improving explanations of code to other developers”. Web site: http://programmers.stackexchange.com/questions/187882/how-can-i-become-better-on-explaining-the-code-to-other-developers

Strachey, C., 2000. “Fundamental Concepts in Programming Languages”, Higher-Order and Symbolic Computation, Vol. 13, pp. 11-49, DOI: 10.1023/A:1010000313106

Teague, D. and Lister, R., 2014. “Programming: Reading, Writing and Reversing, in ITICSE’14 Proc. of 2014 conf. on Innovation & Technology in Computer Science Education, pp. 285-290, ACM. DOI: http://dx.doi.org/10.1145/2591708.2591712

Yagel, R., 2015. “LIDO – Wiki based Living Documentation with Domain Knowledge”, Proc. SKY’2015 6th International Workshop on Software Knowledge, pp. 26-30, DOI: http://dx.doi.org/10.5220/0005643700220026