How to Prepare an API for Programming in Natural Language

Sebastian Weigelt[1], Mathias Landhäußer[2], and Martin Blersch[1]

[1] Karlsruhe Institute of Technology, Am Fasanengarten 5, 76131 Karlsruhe, Germany
{weigelt|blersch}@kit.edu
[2] thingsTHINKING GmbH, Haid-und-Neu-Straße 7, 76131 Karlsruhe, Germany
mathias@thingsTHINKING.net

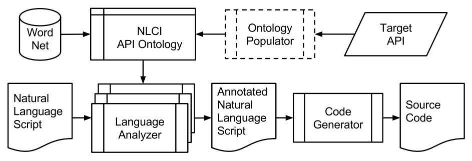
Abstract. Natural language interfaces are becoming more and more common but are extremely difficult to build, to maintain, and to port to new domains. NLCI, the Natural Language Command Interpreter, is an architecture for building and porting such interfaces quickly.
NLCI accepts commands as plain English texts and translates the input sentences into sequences of API calls that implement the intended actions. At its core is an ontology that models the API.
In this demonstration we show how a developer can provide a natural language interface for his or her API by preparing an API ontology. We also show how NLCI analyzes the input text. As an example we use an API that steers a Lego EV3 robot. A short video illustrating the process is available at http://dx.doi.org/10.5445/DIVA/2019-692.

Keywords: Natural language processing · end-user programming.

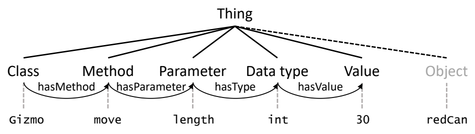
1 Introduction

There is a growing demand for language and speech interfaces and users will soon expect them to be available everywhere. While end-users welcome the simplicity of such interfaces, software engineers face a daunting task: language and speech interfaces are hard to build, improve, and maintain. Building them requires competence in machine learning (ML), natural language processing (NLP), natural language (NL) grammars, and inference engines to map the users' commands to executable code. First attempts for programming in natural language (PNL) with restricted domains were already made in the 1970s [?]; recent works focus on specific environments [?] or tasks [?]. Complementary approaches allow a nearly unlimited domain but restrict the language [?,?]. Others analyzed how non-programmers describe solutions for programming problems [?].

We demonstrate the Natural Language Command Interpreter (NLCI), an architecture that connects end-user APIs to a natural language interface quickly. We describe the process of connecting an API to NLCI to make it programmable



(a) NLCI's analyes process.



(b) The elements of the API ontology.

Fig. 1: NLCI in a nutshell [?].

in natural language. The examples are drawn from a project in which we programmed a Lego EV3 robot [?]. A developer willing to add a language interface to his or her API only needs to provide an ontology of that API. NLCI can then generate API calls from textual commands in unrestricted English.

2 Programming in Natural Language

From an end-user's perspective, NLCI is a translator from English prose to source code. Internally, NLCI splits the translation into various analyses (see Figure 1a). NCLI does not constrain the language of the input but relies on linguistic patterns that its text analyses hinge on. It uses linguistic phenomena and *not* on the purpose, features, or classes of the API and thus the analyses are independent of them. NLCI models the API in an ontology that acts as a bridge from text elements to API elements. Its structure (see Figure 1b for an example) captures the major concepts of object-oriented programming languages and stores linguistic information such as synonyms for class and method names. Consequently, one can implement language analyses without any knowledge about a particular API; the ontology decouples language analysis and the targeted API.

When NLCI analyzes the input given in Figure 2, it first reconstructs the chronological order of the actions. Next, the control structure in the first sentence is analyzed. NLCI recognizes that the action *turn* should be performed until the condition *see the can* holds. In the code this is realized as a do-while loop. Then it identifies the API elements for the actions, agents, and objects. NLCI generates a mapping from *Gizmo* to `Robot` due to the synonym information.

The upper half of Figure 1a shows the setup performed by the developer: The target API must be provided as an API ontology and enriched with linguistic information; WordNet can, e.g., be used to derive synonyms for the API elements. The lower half shows the process in production mode initiated by the end-user. NLCI's language analysis pipeline is split-up into two separate parts: text preparation with standard NLP tools[3] and special modules for natural language understanding (NLU). After the language analyses, all results are available as text annotations. From this NLCI builds an internal structure resembling an abstract syntax tree (AST). A code generator (depending on the target programming language only) uses the AST and the API ontology to produce the desired

[3] Such as Stanford's CoreNLP, see https://stanfordnlp.github.io/CoreNLP/.

Input: Natural Language Script
Gizmo, turn 5 degrees to the right until you see the can.
Before that, move 15 centimeters forward very fast.
Input: API Ontology (Excerpt)
Class: Robot (Synonyms: Gizmo, . . .)
Method: void : Robot.move(Direction: d, Distance: cm, Speed: s)
Method: void : Robot.turn(Direction: d, int: degrees)
Method: boolean : Robot.canSee(Object: o)

Output: Generated Code
robot.move(*Direction.forward*, 15, *Speed.fastest*);
do { robot.turn(*Direction.right*, 5); }
while (not robot.canSee(*can*))

Fig. 2: Input/Output Example: Given the English script in the top, NLCI reorders the described actions, identifies the loop, and maps the text elements to the ontology API elements. Then it generates the desired code.

source code. This design offers two major advantages. First, analyses can draw from previous results, build on them, and refine them. Second, any module can be evaluated separately and improved or replaced if necessary.

As of today, NLCI offers three NLU modules. The first module reconstructs proper time lines so that the actions can be executed in the desired order. Humans tend to describe instructions non-sequentially, e.g., "Do A. Do B. But before that, do C." Generating the method invocation in the textual order does not produce the desired results. NLCI ensures that the method call for C is generated before B. We use keyphrases and structural information to determine the correct order of events. Reference [**?**] gives an in depth description and evaluation of the time line reconstruction.

The second module extracts control structures from the input. For example, a user might say, "Do A three times," or, "Do A. At the same time do B." The first is an implicit description of a loop, the second implies parallelism. To synthesize control structures we use a similar approach as for the time line reconstruction. Besides keyphrases, we use part-of-speech tags. Reference [**?**] describes the approach for control structure extraction as well as its evaluation.

The third module maps text elements to ontology elements and generates method calls. If a user writes, "Gizmo turns to the can," humans naturally understand that there is an agent *Gizmo* that *turns to* (the action) another object, a *can*. The module identifies the respective classes and methods in the API ontology and generates a method call, i.e., in this example `Gizmo.turnTo(can)`. To produce method calls, we first transform the textual input into an intermediate predicate-like representation: *action(agent, object[])*. Each element consists of one or more words from the input. To obtain a predicate we analyze part-of-speech tags, parse trees, and dependency graphs. The predicates abstract from the grammatical structure of the input. For example, passive and active voice versions of the same sentence result in identical predicates; attributes expressed as either an adjective or in a subordinate clause are represented as the same parameter. Finally, we map the elements of the predicate to ontology individuals. Our approach creates candidates, scores them, and finally selects one mapping per predicate. We identify the best call sequence for given the input and optimize the score of method calls globally. Reference [?] describes the API mapping approach along with a comprehensive evaluation.

When using NLCI in a specific domain, e.g., spreadsheet calculations, a developer can increase NLCI's performance with specialized analyses. They can be integrated in NLCI's pipeline at any stage to improve intermediate information or to refine results.

3 Preparing an API for NLCI

A developer willing to connect his or her end-user API to NLCI needs to supply NLCI with an API ontology. NLCI specifies the layout of the ontology (see Figure 1b). Consequently, NLCI's language analyses can be used with any object-oriented API as long as the corresponding ontology is set up properly. It defines the following ontology concepts and the relations between them: class, method, parameter, data type, and value. This structure may be extended, e.g., to accommodate special API features or programming language elements.

Since the linguistic analyses hinge natural language words, the ontology also must contain a natural language representation of the API. This means that the ontology does not only contain the technical information about the API but a description of the API in English. The ontology stores the identifiers used in the API in a tokenized form. For example, class names such as `Ev3RobotGizmo` must be split into the individual words `EV3`, `robot` and `Gizmo`. To cover a broader spectrum of language variations, the ontology also stores synonyms. For example, we could add `bot`, `machine`, and `droid` to augment the NL description of `Ev3RobotGizmo`. If the API does not use descriptive names, the API developer must provide them. Given an API with descriptive names that follows naming conventions consistently (e.g., camel case), NLCI can tokenize the identifiers automatically; synonyms can be harvested from lexical databases such as WordNet.

The API ontology can either be populated manually, automatically, or with a hybrid approach. Manual population is straightforward but laborious. Therefore, it is advisable to implement an ontology populator. As reading the API and pushing the information into the ontology does not depend on the API itself but on the input files, an API developer needs only one populator per programming language or input format. Ontology populators can be built with little effort: Ours for Java uses an open-source Java parser and has only 436 lines of code. However, it might be necessary to add manual steps to the automatic extraction: if the developer wants to provide elements that are not delivered with the API (e.g., external objects), he or she has to add them manually to the ontology.

Given the API ontology, NLCI can analyze NL scripts for every end-user API. The final module to supply is a code generateor. It inherently depends on the programming language that is used with the API. The code generator can use the AST-like representation of the user input that is created at the end of NLCI's language analysis process. Building a code generator is straightforward and does not depend on a particular API: a Java code generator can be used with any Java API.

4 Conclusion and Future Work

In this demonstration we presented NLCI, an architecture to build natural language interfaces for object-oriented APIs. NLCI analyzes unrestricted English texts and generates coherent API calls from prose and imperative sentences.

When using NLCI with an API, the API developer has to configure NLCI with an abstract model of the API in the form of an ontology. The API ontology helps NLCI to bridge the linguistic gap between the NL input and the API. Generating such an API ontology is easy: e.g., a simple generator for Java APIs can be implemented in less than 500 lines of Java code. NLCI supports the ontology generation process with tools that preprocess and enrich an ontology as long as the underlying API uses descriptive names and follows common naming conventions. Therefore, API developers can focus on building useful APIs instead of dealing with NLU problems.

NLCI offers a domain independent set of analyses that handle most NL scripts for end-user programming APIs sufficiently. Code generation depends only on the targeted programming language and thus is usable with any API.

The biggest challenge in meeting the end-users' needs is addressing spoken language. As of today, NLCI relies on syntactical analyses that do not perform well on ungrammatical phrases or speech. To address this issue, we are investigating how to rely less on syntactical information or to detect and to recover from NLP errors [**?**].

References

1. Ballard, B.W., Biermann, A.W.: Programming in Natural Language: "NLC" As a Prototype. In: Proceedings of the 1979 Annual Conference (ACM). ACM (1979)
2. Landhäußer, M., Hey, T., Tichy, W.F.: Deriving Timelines from Texts. In: 3rd Int. Wksp. on Realizing Artificial Intelligence Synergies in Software Engineering (2014)
3. Landhäußer, M., Hug, R.: Text Understanding for Programming in Natural Language: Control Structures. In: 4th Int. Wksp. on Realizing Artificial Intelligence Synergies in Software Engineering (2015)
4. Landhäußer, M., Weigelt, S., Blersch, M.: Teaching Research Methodologies with a Robot in a CS Lab Course. In: 8th Int. Conf. on Robotics in Education. Ed.: R. Balogh (2017)
5. Landhäußer, M., Weigelt, S., Tichy, W.F.: NLCI: A Natural Language Command Interpreter. Automated Software Engineering (2016)
6. Le, V., Gulwani, S., Su, Z.: Smartsynth: Synthesizing Smartphone Automation Scripts from Natural Language. In: MobSys'13. vol. 2 (2013)
7. Liu, H., Lieberman, H.: Metafor: Visualizing Stories as Code. In: 10th Int. Conf. on Intelligent User Interfaces. ACM (2005)
8. Pane, J.F., Ratanamahatana, C., Myers, B.A.: Studying the Language and Structure in Non-programmers' Solutions to Programming Problems. Int. Journal of Human-Computer Studies **54**(2) (2001)
9. Price, D., Riloff, E., Zachary, J., Harvey, B.: NaturalJava: A Natural Language Interface for Programming in Java. In: 5th Int. Conf. on Intelligent User Interfaces. ACM (2000)

10. Thummalapenta, S., Devaki, P., Sinha, S., Chandra, S., Gnanasundaram, S., Nagaraj, D., Kumar, S., Kumar, S.: Efficient and Change-resilient Test Automation: An Industrial Case Study. In: 35th Int. Conf. on Software Engineering (2013)
11. Weigelt, S., Tichy, W.F.: Poster: ProNat: An Agent-Based System Design for Programming in Spoken Natural Language. In: 37th IEEE Int. Conf. on Software Engineering (ICSE). vol. 2 (2015)