Jan Jürjens, Kurt Schneider (Hrsg.): Software Engineering 2017,
Lecture Notes in Informatics (LNI), Gesellschaft für Informatik, Bonn 2017 139

NLCI – A Natural Language Command Interpreter

Mathias Landhäußer[1] Sebastian Weigelt Walter F. Tichy

Keywords: Programming in Natural Language, End-user Programming, Knowledge-based Software Engineering, Program Synthesis, Natural Language Processing for Software Engineering

Introduction Natural language (NL) interfaces are becoming more and more common, because they are powerful and easy to use. However, such interfaces are extremely difficult to build, to maintain, and to port to new domains. They involve competence in speech and natural language processing (NLP), NL grammars, and inference engines that map the input to whatever the application requires.

We present an approach for building and porting such interfaces quickly. NLCI is a natural language command interpreter that accepts action commands in English and translates them into executable code. The core component is an ontology that models an API. Construction of the ontology can be automated if the API uses descriptive names for its components. In that case, the language interface can be generated completely automatically.

The interface can be used, for instance, for instructing robots, programming home automation systems, manipulating spreadsheets, controlling games, or for working with any API that is suitable for end-user programming.

The important advance reported in this paper is that interfaces that work with unrestricted English text require only a domain ontology to be built and no other expertise. The ontology can even be generated automatically, if the API has certain properties. Our approach is a first step on the road to simplify the construction of next-generation user interfaces.

Approach Processes that translate NL into source code usually either target a specific domain, restrict the input language, or both. Our language analysis is completely domain agnostic. The domain knowledge is stored in an ontology and loaded before processing the input. All information derived is annotated in the input text and the (necessarily platform specific) code generation engine can make use of it without knowing anything about NL.

Translation Process Fig. 1 illustrates the overall process: First we populate the ontology that contains the domain specifics (i.e. the API with all classes and their methods); to allow for fuzzy language matching, we enrich the API with synonyms from WordNet. The domain ontology must be built only once per API and can be easily extended. Given an input script, we parse it, enrich it with structural information (such as control structures [LH15]), and identify actors, actions, and (grammatical) objects. For every sentence we identify the classes and methods to invoke (including parameters). Before handing over

[1] Karlsruhe Institute of Technology, Institute for Program Structures and Data Organization, Am Fasanengarten 5, 76131 Karlsruhe, Germany, {landhaeusser | weigelt | tichy}@kit.edu, https://ps.ipd.kit.edu

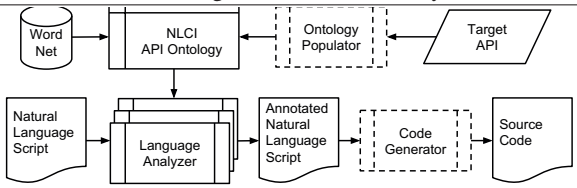

Fig. 1: The NLCI architecture. The ontology populator and the code generator depend on the programming language only; all other components depend on the natural language only.

the annotated text to the code generator, NLCI checks and and corrects the sequential order of the script [LHT14]. The last phase generates actual code for the target programming language. As this step is programming language dependent, one must provide a code generator for each programming language one wants to support. For an in-depth description of NLCI see references [LWT16] and [La16].

Evaluation NLCI has been tested on two radically different domains: openHAB, an API for home automation, and Alice, a programming environment for building 3D animations. Both of the APIs were ontologized automatically and both were tested with benchmarks of scripts. For each script, a gold standard solution was constructed by hand.

In summary, NLCI produces the correct API calls 67% of the time, with a precision of 78%. Of course, this is not good enough for practical use. Naturally, future work will have to improve both precision and recall, which means that both the parser and the matching component need to be improved significantly.

The results are promising and show that interpreting commands stated in NL in these two domains is feasible, yet we need to improve the accuracy of our matching algorithm. Though NLCI is limited to written input at the moment, future work will use a speech front-end to generate text for processing by NLCI [WT15].

References

[La16] Landhäußer, M.: Eine Architektur Für Programmsynthese Aus Natürlicher Sprache. KIT Scientific Publishing, Karlsruhe, 2016.

[LH15] Landhäußer, M.; Hug, R.: Text Understanding for Programming in Natural Language: Control Structures. In: Proc. of the 4th Int. Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. May 2015.

[LHT14] Landhäußer, M.; Hey, T.; Tichy, W. F.: Deriving Timelines from Texts. In: Proc. of the 3rd Int. Workshop on Realizing Artificial Intelligence Synergies in Software Engineering. pp. 45–51, June 2014.

[LWT16] Landhäußer, M.; Weigelt, S.; Tichy, W. F.: NLCI: A Natural Language Command Interpreter. Automated Software Engineering, August 2016.

[WT15] Weigelt, S.; Tichy, W. F.: Poster: ProNat: An Agent-Based System Design for Programming in Spoken Natural Language. In: 2015 IEEE/ACM 37th IEEE Int. Conf. on Software Engineering (ICSE). volume 2, pp. 819–820, May 2015.