Programming in Natural Language with *fu*SE: Synthesizing Methods from Spoken Utterances Using Deep Natural Language Understanding

Sebastian Weigelt, Vanessa Steurer, Tobias Hey, and Walter F. Tichy
Karlsruhe Institute of Technology
Institute for Program Structures and Data Organization
Karlsruhe, Germany
weigelt@kit.edu, vanessa.steurer@web.de,
hey@kit.edu, tichy@kit.edu

Abstract

The key to effortless end-user programming is natural language. We examine how to teach intelligent systems new functions, expressed in natural language. As a first step, we collected 3168 samples of teaching efforts in plain English. Then we built *fu*SE, a novel system that translates English function descriptions into code. Our approach is three-tiered and each task is evaluated separately. We first classify whether an intent to teach new functionality is present in the utterance (accuracy: 97.7% using BERT). Then we analyze the linguistic structure and construct a semantic model (accuracy: 97.6% using a BiLSTM). Finally, we synthesize the signature of the method, map the intermediate steps (instructions in the method body) to API calls and inject control structures (F_1: 67.0% with information retrieval and knowledge-based methods). In an end-to-end evaluation on an unseen dataset *fu*SE synthesized 84.6% of the method signatures and 79.2% of the API calls correctly.

1 Introduction

Intelligent systems became rather smart lately. One easily arranges appointments by talking to a virtual assistant or controls a smart home through a conversational interface. Instructing a humanoid robot in this way no longer seems to be futuristic. For the time being, users can only access built-in functionality. However, they will soon expect to add new functionality themselves. For humans, the most natural way to communicate is by natural language. Thus, future intelligent systems must be programmable in everyday language.

Today's systems that claim to offer programming in natural language enable laypersons to issue single commands or construct short scripts (e.g. Mihalcea et al. (2006); Rabinovich et al. (2017)); usually no new functionality is learned. Only a few addressed learning new functionality from natural language instructions (e.g. Le et al. (2013); Markievicz et al. (2017)). However, even recent approaches still either restrict the language or are (over-)fitted to a certain domain or application.

We propose to apply deep natural language understanding to the task of synthesizing methods from spoken utterances. Our approach combines modern machine learning techniques with information retrieval and knowledge-based methods to grasp the user's intent. As a first step, we have performed a user study to investigate how laypersons teach new functionality with nothing but natural language. In a second step, we develop *fu*SE (Function Synthesis Executor). *fu*SE translates teaching efforts into code. On the basis of the gathered data we constructed a three-tiered approach. We first determine, whether an utterance comprises an explicitly stated intent to teach a new skill. Then, we decompose these teaching efforts into distinct semantic parts. We synthesize methods by transferring these semantic parts into a model that represents the structure of method definitions. Finally, we construct signatures, map instructions of the body to API calls, and inject control structures.

2 Related Work

The objective of programming in natural language was approached from different perspectives over the years. Quite a few approaches are natural language interfaces to code editors (Price et al., 2000; Begel, 2004; Begel and Graham, 2005; Désilets et al., 2006). However, they assume that users literally dictate source code. Thus, these approaches are intended for developers rather than laypersons. Other approaches such as *Voxelurn* by Wang et al. (2017) aim to naturalize programming languages to lower the hurdle for programming novices.

Approaches for end-user programming in natu-

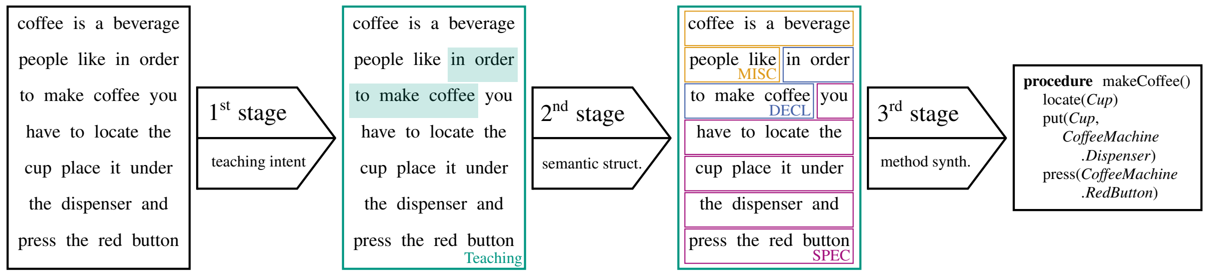

Figure 1: Schematic overview of *fu*SE's three-tiered approach.

ral language take up the challenge of bridging the semantic gap between informal spoken or written descriptions in everyday language and formal programming languages. Early systems were syntax-based (Winograd, 1972; Ballard and Biermann, 1979; Biermann and Ballard, 1980; Biermann et al., 1983; Liu and Lieberman, 2005). Some were already capable to synthesize short scripts including control structures and comments, e.g. *NLP for NLP* by Mihalcea et al. (2006). Others take the user in the loop and create scripts with a dialog-driven approach (Le et al., 2013). In further developments intelligent assistants offer their service to assist with programming (Azaria et al., 2016). Often these assistants support multi-modal input, e.g. voice and gestures (Campagna et al., 2017, 2019). Others combine programming in natural language with other forms of end-user programming, such as programming by example (Manshadi et al., 2013) or programming by demonstration (Li et al., 2018).

Some authors such as Landhäußer et al. (2017) and Atzeni and Atzori (2018a,b) take a knowledge-based approach by integrating domain and environmental information in the form of ontologies.

Suhr and Artzi (2018) employ a neural network to learn a situational context model that integrates the system environment and the human-system-interaction, i.e. the dialog. Many recent approaches integrate semantic parsing in the transformation process (Guu et al., 2017; Rabinovich et al., 2017; Chen et al., 2018; Dong and Lapata, 2018). Even though the natural language understanding capabilities are often impressive, the synthesized scripts are still (semantically) erroneous in most cases. Additionally, learning of new functionality is not covered by approaches of that category so far.

Programming in natural language is of particular interest in the domain of humanoid robotics (Lauria et al., 2001, 2002; She et al., 2014; Mei et al., 2016). People expect to teach them as they teach human co-workers. Therefore, some authors, e.g. Markievicz et al. (2017), use task descriptions that were intended to instruct humans to benchmark their approach. However, often the assumed vocabulary is rather technical (Lincoln and Veres, 2012). Thus, the usability for laypersons is limited.

3 Approach

The goal of our work is to provide a system for programming in (spoken) natural language. Laypersons shall be enabled to create new functionality in terms of method definitions by using natural language only. We offer a general approach, i.e. we do not restrict the natural language regarding wording and length. Since spontaneous language often comprises grammatical flaws, disfluencies, and alike, our work must be resilient to these issues.

We decompose the task in three consecutive steps. The rationale behind this decision is as follows. On the one hand, we can implement more focused (and precise) approaches for each task, e.g. using machine learning for one and information retrieval for another. On the other hand, we are able to evaluate and optimize each approach individually. The stages of our three-tiered approach are the following (see Figure 1 for an example):

1. **Classification of teaching efforts:** Determine whether an utterance comprises an explicitly stated teaching intent or not.

2. **Classification of the semantic structure:** Analyze (and label) the semantic parts of a teaching sequence. Teaching sequences are composed of a *declarative* and a *specifying* part as well as superfluous information.

3. **Method synthesis:** Build a model that represents the structure of methods from syntactic information and classification results. Then, map the actions of the specifying part to API calls and inject control structures to form the body; synthesize the method signature.

The first two stages are classification problems. Thus, we apply various machine learning techniques. The first stage is a sequence-to-single-label task, while the second is a typical sequence-to-sequence task. For the first we compare classical machine learning techniques, such as logistic regression and support vector machines, with neural network approaches including the pre-trained language model *BERT* (Devlin et al., 2019). For the second task we narrow down to neural networks and BERT. A more detailed description of the first two stages may be found in (Weigelt et al., 2020). The implementation of the third stage is a combination of syntactic analysis, knowledge-based techniques and information retrieval. We use semantic role labeling, coreference analysis, and a context model (Weigelt et al., 2017) to infer the semantic model. Afterwards, we synthesize method signatures heuristically and map instructions from the body to API calls using ontology search methods and datatype analysis. Additionally, we inject control structures, which we infer from keywords and syntactic structures. To cope with spontaneous (spoken) language, our approach relies on shallow NLP techniques only.

3.1 Dataset

We carried out a study to examine how laypersons teach new functionality to intelligent systems. The study consists of four scenarios in which a humanoid robot should be taught a new skill: greeting someone, preparing coffee, serving drinks, and setting a table for two. All scenarios take place in a kitchen setting but involve different objects and actions. Subjects were supposed to teach the robot using nothing but natural language descriptions. We told the subjects that a description ideally comprises a declaration of intent to teach a new skill, a name for the skill, and an explanation of intermediate steps. However, we do not force the subjects into predefined wording or sentence structure. Instead, we encouraged them to vary the wording and to 'speak' freely. We also instructed them to imagine that they were standing next to the robot. After the short introduction, we successively presented the scenarios to the subjects. Finally, we requested some personal information in a short questionnaire.

We used the online micro-tasking platform *Prolific*[1,2]. In less than three days, 870 participants completed the study. The share of male and female participants is almost equal (50.5% vs. 49.5%); more than 60% are native English speakers. Most of them (70%) had no programming experience at all. An analysis of the dataset revealed that there is barely any difference in the language used by subjects, who are inexperienced in programming, compared to more experienced subjects (except for a few subjects that used a rather technical language). The age of the participants ranges from 18 to 76 with more than half being 30 or younger.

	desc.	w. (total)	w. (unique)
sc. 1 (greet)	795	18,205	566
sc. 2 (coffee)	794	26,005	625
sc. 3 (drinks)	794	33,001	693
sc. 4 (table)	785	31,797	685
total	3,168	109,008	1,469

Table 1: The number of descriptions, words, and unique words per scenario and in the entire dataset.

The collected data comprises 3,168 descriptions with more than 109,000 words altogether (1,469 unique words); the dataset statistics are depicted in Table 1. We provide a set of six descriptions from the dataset in Table 13 (Appendix A). A thorough analysis of the dataset revealed that a notable share (37%) lacks an explicitly stated intent to teach a skill, albeit we even consider phrases such as "to prepare lunch" as teaching intent. Regarding the semantic structure, we observed that the distinct parts can be clearly separated in almost all cases. However, the respective parts occurred in varying order and are frequently non-continuous.

The data was jointly labeled by two of the authors. We first attached the binary labels *teaching* and *non-teaching*. These labels correspond to the classification task from the first stage. Then we add ternary labels (*declaration*, *specification*, and *miscellaneous*) to all words in descriptions that were classified as teaching effort in the first step. This label set is used for the second stage. The distribution of the labels is depicted in Table 2.

Both label sets are unequally distributed, which may cause the machine learning models to overfit in favor of the dominating label. This mainly affects the ternary classification task; the

[1]Prolific: `https://www.prolific.co/`

[2]We decided to gather textual responses, even though speech recordings would be more natural. However, from previous studies we learned that subjects more willingly write texts than speak. Besides, the audio quality of recordings is often poor, when subjects use ordinary microphones.

binary			ternary			
teaching	non-teaching	total	declaration	specification	miscellaneous	total
1,998 (.63)	1,170 (.37)	3,168	15,559 (.21)	57,156 (.76)	2,219 (.03)	74,934

Table 2: The distribution of binary and ternary labels in the dataset. The resp. share is given in parenthesis.

	random	scenario
Decision Tree	(.893) .903	(.861) **.719**
Random Forest	(.917) .909	(**.893**) .374
SVM	(.848) .861	(.870) .426
Naïve Bayes	(.771) .801	(.765) .300
Logistic Regression	(**.927**) **.947**	(.891) **.719**
baseline (ZeroR)	.573	.547

Table 3: Classification accuracy achieved by classical machine learning techniques on validation (in paren.) and test set. The best results are printed in bold type.

label *specification* distinctly dominates (76%) the others. The entire dataset is publicly accessible (open access), including raw data, labeled data, meta-data, and scenario descriptions: `http://dx.doi.org/10.21227/zecn-6c61`.

3.2 First Stage: Teaching Intents

The first step of *fu*SE is discovering teaching intents in utterances. An utterance can either be an effort to teach new functionality or merely a description of a sequence of actions. This problem is a typical sequence-to-single-label task, where the words of the utterance are the sequential input and the output is either *teaching* or *non-teaching*.

To train, validate, and test classifiers we split up the dataset in two ways. The first is the common approach to randomly split the set in an 80-to-20 ratio, where 80% of the data is used for training and 20% for testing. As usual, we again split the training set in 80 parts for training and 20 for validation. However, we felt that this approach does not reflect realistic set-ups, where a model is learned from historical data and then applied to new unseen data, that is semantically related but (potentially) different. Therefore, we introduced an additional so-called *scenario-based* split in which we separate the data according to the scenarios. We use three of the four scenarios for training and the remaining for testing. Note that we again use an 80-20 split to divide training and validation sets.

We applied classical machine learning and neural network approaches to the task. The classical techniques are: decision trees, random forests, support vector machines, logistic regression, and Naïve Bayes. As baseline for the classification accuracy we use the so-called *Zero-Rule* classifier (ZeroR); it always predicts the majority class of the training set, i.e. *teaching* in this case.

We transform the words to bag-of-words vectors and use tri- and quadrigrams as additional features. The measured accuracy of each classifier on the random and scenario-based data is depicted in Table 3; the validation set accuracy is given in parenthesis and the test set accuracy without.

On the random set all classifiers exceed the baseline. Thus, the (slightly) imbalanced dataset does not seem to affect the classifiers much. Logistic regression performs surprisingly well. However, on the scenario-based split the accuracy of all classifiers decreases drastically. While the accuracies on the validation set remain stable, these classifier techniques are unable to generalize to unseen input. The logistic regression remains the best classifier. However, its accuracy decreases to 71.9%.

These results reinforced our intuition that deep learning is more appropriate for this task. We implemented a broad range of neural network architectures: artificial neural networks, convolutional networks, and recurrent networks, including LSTMs and GRUs and their bidirectional variants. We experimented with additional layers, which we systematically added to the networks, such as dropout (DO), dense (D), or global max pooling (GMax). We altered all hyper-parameters in reasonable ranges of values[3]. We present only the best performing configurations, i.e. architecture and hyper-parameter combinations, in Table 4. Detailed information on the tested hyper-parameter values and further results may be found in Appendices B and C. The words from the input are represented as *fastText* word embeddings (Bojanowski et al., 2017; Joulin et al., 2017); we use the 300-dimensional embeddings that were trained on the *Common Crawl* dataset[4] by *Facebook Research* (Mikolov et al.,

[3]Note that we do not discuss the influence of varying epoch numbers, since we used *early stopping*, i.e. the training stops when the validation loss stops decreasing.

[4]Common Crawl: `https://commoncrawl.org/`

network architecture	random	scenario
C(128,3), Max(2), C(64,3), GMax, D(10)	(.952) **.971**	(.962) .874
C(128,5), Max(2), C(128,5), GMax, D(10)	(.954) .966	(**.977**) .862
BiGRU(32), DO(.2), D(64), DO(.2)	(.952) .959	(.958) **.932**
BiLSTM(128), D(64)	(**.956**) .959	(.962) .919
BERT, 5 epochs	(.973) .981	(.991) .969
BERT, 10 epochs	(**.976**) **.982**	(**.992**) .973
BERT, 300 epochs	(.962) **.982**	(**.992**) **.977**
baseline (Log. Reg.)	(.927) .947	(.891) .719

Table 4: Classification accuracy for neural networks on validation (in parenthesis) and test set (best in bold).

2018). Moreover, we use Google's pre-trained language model BERT (base-uncased), which we equipped with a flat binary output layer.

The results attest that deep learning approaches clearly outperform the best classical technique (logistic regression). In particular, the accuracies show smaller differences between random and scenario-based split. This suggests that the classification is more robust. The best accuracy on the scenario test set is achieved by a bidirectional GRU: 93.2%. Using BERT, the accuracy increases by more than 4% with a peak at 97.7% using 300 training epochs. However, the ten-epochs version is a feasible choice, since the accuracy loss is negligible and the training savings are immense.

3.3 Second Stage: Semantic Structures

The second stage, detecting the semantic parts in teaching efforts, is a typical sequence-to-sequence-labeling task with the labels *declaration*, *specification*, and *miscellaneous*. Even though these semantic structures correspond to phrases from a grammatical point of view, we decided to use per-word labels. For this task we only use neural network approaches and BERT. The remaining set-up is similar to the first stage. We again use fastText embeddings and vary the network architectures and hyper-parameters. Except for a ternary output layer, we use the same configuration for BERT as in the first stage.

The results for both, the random and scenario-based split, are reported in Table 5[5]. The bidirectional architectures – be it GRU or LSTM – are

network architecture	random	scenario
BiLSTM(128)	(.987) .985	(.981) **.976**
BiGRU(128)	(.985) .985	(**.982**) .968
BiLSTM(128), DO(.2)	(**.988**) **.988**	(.981) .975
BiLSTM(256), DO(.2)	(.987) .985	(**.982**) .975
BERT, 5 epochs	(.979) .982	(.979) .965
BERT, 10 epochs	(**.983**) **.985**	(.983) .972
BERT, 300 epochs	(.981) .983	(**.985**) **.973**
baseline (ZeroR)	.759	.757

Table 5: Classification accuracy achieved by neural networks on validation (in parenthesis) and test set for the second stage. The best results are printed in bold type.

the clear choice for this task; accuracy values are consistently high. Most encouragingly, the decline on the scenario data is negligible (less than 1%). Apparently, the models generalize well and are thus resilient to a change in vocabulary. For the second stage the use of BERT is of no advantage; the results even fall behind the best RNN configurations.

3.4 Third Stage: Method Synthesis

During stage three we first transfer the natural language utterances into a model that represents both method definitions and scripts. Afterwards, we synthesize methods (or scripts) from this model. We create a method signature and map instructions in the body to API calls; to synthesize scripts we only map the instructions and inject control structures.

Before we can transfer natural language utterances to the semantic model we must perform a few NLP pre-processing steps that enrich the input with syntactic and semantic information. To obtain parts of speech (PoS), we apply a joint tagging approach; we consolidate the PoS tags produced by the *Stanford Log-linear Part-Of-Speech Tagger* (Toutanova et al., 2003) and *SENNA* (Collobert et al., 2011). The Stanford Tagger also provides us with word lemmas. Then we detect individual events in terms of clauses. Since our approach is supposed to cope with spoken language, we are unable to make use of punctuation. Instead, we split the input in a continuous sequence of instructions based on heuristics that make use of PoS tags and keywords. However, the instructions do not necessarily span complete clauses. Thus, we can not apply common parsers. Instead, we use the shallow parser *BIOS*[6] that provides us with chunks. To obtain semantic roles for each instruction, we again

[5] Again, we only present the best configurations here. For more configurations, refer to Table 16 in Appendix C.

[6] `http://www.surdeanu.info/mihai/bios/`

class	description
Thing	Top concept of the ontology
└ System	(Sub-)Systems (API classes)
└ Method	System functions (API methods)
└ Parameter	Parameter names
└ DataType	Data types used by the system, e.g., *int* or *Graspable*
└ Object	External objects [empty here]
└ State	Object states [empty here]

Table 6: Domain ontology structure for systems.

class	description
Thing	Top concept of the ontology
└ Object	Objects in environment
└ Graspable	Graspable objects, e.g., *cup*
└ Openable	Openable objects, e.g., *bottle*
. . .	
└ State	Object states, e.g., *opened*

Table 7: Domain ontology structure for environments.

employ SENNA[7]. Word senses are disambiguated using the tool *Babelfy* (Moro et al., 2014). Since Babelfy is linked to *WordNet* (Fellbaum, 1998), we can also make use of synonyms.

We use ontologies to model the target systems, i.e. APIs. An ontology represents the classes, methods, parameters, data types, and values (resp. value ranges), of an API (similar to the ontologies used by Landhäußer et al. (2017) and Atzeni and Atzori (2018a,b)). The basic ontology structure is depicted in Table 6. If the system is supposed to interact with an environment, we employ additional ontologies that model the environment including objects and their states (see Table 7). Environment ontologies are merged into system ontologies by copying concepts to the respective placeholders.

To bridge the semantic gap between natural and programming language we introduce a semantic model, as depicted in Figure 2. The model resembles the basic structure of method definitions. However, the leaves are composed of natural language phrases. To determine the phrases that will make up the model elements, we first smooth the classification results provided by the second stage. *fu*SE maps all phrases of an instruction to the same second-level model element, i.e. either method signature or an instruction of the body. Therefore, we

[7]SENNA uses the semantic role label set defined in the CoNLL-2004 resp. CoNLL-2005 shared tasks (Carreras and Màrquez, 2004, 2005).

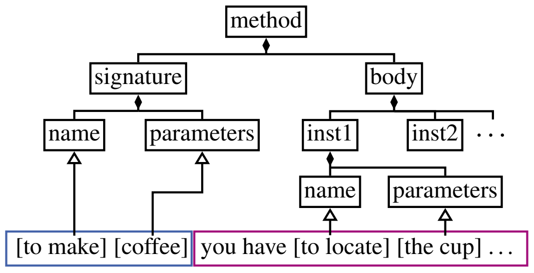

Figure 2: Exemplary semantic model for an utterance.

unify the second stage classification labels for each instruction using majority decision. Afterwards, we map phrases to leaf elements. Roughly speaking, we use the roles provided by semantic role labeling (SRL) and map predicates to names and arguments to parameters. If we detect a coreference, we substitute the referring expression with the referent, e.g. *it* with *the cup*. We also add a lemmatized variant of the phrase and all synonyms. Note that the `parameters` are a list of phrases.

The first step to create method definitions is signature synthesis. To construct a meaningful name, we heuristically clean up the phrase, e.g. remove auxiliary verbs and stop words, and concatenate the remaining words. The parameters are either mapped to data types to infer formal parameters or – if no mapping is to be found – they are attached to the name. For instance, assuming that the declarative instruction is *serving wine means*, *fu*SE extracts *serve* as the first part of the name. Then it tries to map *wine* to an ontology individual (as discussed later). Assuming it finds the individual `RedWineBottle` and it is an instance of the concept `Graspable` in the environment ontology. If the system ontology supports the data type `Graspable`, *fu*SE synthesizes the signature `serve(serve.what : Graspable)`. Otherwise, the method signature `serveWine()` is created.

The instructions in the method body are mapped to API calls. Therefore, we first query the ontologies for each leaf element individually. For the queries we use three sets of words we create from the original phrase, the lemmatized version, and the synonyms. We then build the power sets and all permutations of each set, before we concatenate the words to construct a query set. For instance, for the phrase *is closed*, we produce the query strings: *isclosed, closedis, beclose, closebe, closed, is, . . .* The ontology search returns all individuals with a Jaro-Winkler score (Winkler, 1990) above .4 or

	individuals				API calls			
	pre.	recall	F_1	avg. rank	pre.	recall	F_1	avg. rank
sc. 1	.763	.584 (.776)	.662 (.769)	1.31	.583	.461 (.614)	.515 (.598)	1.47
sc. 2	.783	.742 (.857)	.762 (.818)	1.16	.674	.620 (.713)	.646 (.693)	1.19
sc. 3	.847	.813 (.893)	.830 (.870)	1.16	.672	.645 (.708)	.658 (.690)	1.20
total	.807	.731 (.854)	.767 (.830)	1.20	.653	.590 (.689)	.620 (.670)	1.22

Table 8: The results of the evaluation of the API call mapping for individual elements, i.e. names and parameters, and entire calls. The values in parenthesis denote the results obtained excluding SRL errors.

	total	teach	non-teach	API calls
sc. 1	25	18	7	77
sc. 2	25	19	6	97
sc. 3	25	15	10	123
total	75	52	23	297

Table 9: The dataset used to evaluate the third stage.

a fuzzy score[8] above .15. We decided for these comparatively low thresholds, since we see them as lightweight filters that let pass numerous generally valid candidates. Since an individual may be returned more than once with different scores, we set the score of the individual to the maximum of each of its scores. Afterwards, we construct API calls from the model structure and rate each candidate. We start with the method name candidates. For each candidate we query the ontology for formal parameters. Then, we try to satisfy the parameters with the candidates returned by the individual ontology search. Note that we perform type checking for the parameters (including inheritance if applicable). For instance, for the instruction *take the cup* we may have found the individual `grasp` as candidate for a method name and the parameter candidates `Mug` (type `Graspable`) and `Cupboard` (type `Location`). The ontology indicates that the method `grasp` has one parameter of type `Graspable`. Then, the type check ensures that *fu*SE creates the call candidate `grasp(Mug)` but not `grasp(Cupboard)`. The score is composed of the individual scores of the method names and parameters, the share of mapped words of query string to all words in the query, the ratio of mapped parameters to (expected) formal parameters, and the number of additional (superfluous) parameters. In Appendix D we give a more formal introduction to our scoring approach.

[8]https://commons.apache.org/proper/commons-text/apidocs/org/apache/commons/text/similarity/FuzzyScore.html

The result of the scoring process is a ranked list of candidates for each instruction. For the time being, we simply use the top-ranked candidates to synthesize the method body. However, re-ranking the candidates based on other semantic resources is promising future work. In a last step, we inject control structures, i.e. conditional branching, various types of loops, and concurrency (Weigelt et al., 2018b,c). The approach is rule-based. We use key phrases, such as *in case, until*, and *at the same time*. Proceeding from these anchor points we look for structures that fit into the respective control structure. Here, we apply heuristics on the syntax (based on PoS tags and chunks) and coreference. Utterances that were labeled as *non-teaching* in the first stage also run through the third stage, except for signature synthesis. Thus, we only construct scripts for this type of utterances.

We determine the quality of the approach for the third stage based on utterances from scenarios one, two, and three, since we used scenario four during development. The assessment is partly manual. Hence, we randomly drew 25 utterances from each scenario to reduce the effort. For each description we used the manual labels of first-stage and second-stage classifications and prepared a gold standard for API calls in the method body. Table 9 depicts the dataset. We did not prepare solutions for the signatures, since plenty of valid solutions are imaginable. Thus, we decided to review the signatures manually afterwards. Of the 52 synthesized method names we assessed eight inappropriate. A name is inappropriate if either the name is off-topic or it contains unrelated terms, e.g. `askSpeaker` or `prepareCoffeeFriend` for the scenario *How to prepare coffee*. Moreover, *fu*SE correctly mapped 23 parameters without any false positive.

The API ontology used in our setting (household robot) comprises 92 methods, 59 parameters, and 20 data types. To represent the environment (a kitchen) of the robot, we used another ontology

	individuals				API calls			
	pre.	recall	F_1	avg. rank	pre.	recall	F_1	avg. rank
sc. 5	.823	.854 (.944)	.839 (.879)	1.04	.589	.649 (.722)	.617 (.649)	1.04
sc. 6	.920	.876 (.929)	.898 (.925)	1.06	.711	.679 (.721)	.695 (.716)	1.11
total	.886	.869 (.934)	.877 (.909)	1.05	.668	.670 (.721)	.669 (.694)	1.10

Table 10: The results of the end-to-end evaluation, divided in individual elements, i.e. names and parameters, and entire calls. The values in parenthesis denote the results obtained excluding SRL errors.

	total	teach	non-teach	API calls
sc. 5	50	44	6	158
sc. 6	50	34	16	315
total	100	78	22	473

Table 11: The end-to-end evaluation dataset.

with 70 objects of six types, and six states. Table 8 details the results for the method body synthesis. Besides precision, recall, and F_1, it shows the average rank at which the correct element is to be found. Since the semantic role labeling introduces a vast amount of errors on spoken utterances and our approach heavily depends on it, we also determine recall and F_1 excluding SRL errors. The results are encouraging. We achieve an F_1 value of 76.7% for the individuals and 62.0% for entire calls; in both cases the precision is slightly ahead of the recall. If we excluded SRL errors, the overall performance increases (about 7% for individuals and 5% for calls). Besides the SRL, missing and inappropriate synonyms are a major source of errors. If WordNet lacks a synonym for an important word in the utterance, *fu*SE's API mapping may be unable to determine the correct ontology individual. Contrary, if WordNet provides an inappropriate synonym, *fu*SE may produce an incorrect (superfluous) mapping. In other cases, our language model is unable to capture the semantics of the utterance properly. For example, *fu*SE creates two method calls for the phrase "make sure you close it" : `close(...)` and `make(...)`. It may also produce superfluous mappings for explanatory phrases, such as "the machine fills cups", if the second stage did not classify it as *miscellaneous*. Regarding the composition of API calls (methods plus arguments), the majority of errors is introduced by the arguments. In addition to the afore-mentioned error sources, arguments are often ambiguous. For instance, the phrase "open the door" leaves it up to interpretation, which door was intended to be opened. Even though *fu*SE makes use of an elaborated context model, some ambiguities are impossible to resolve (see section 5). A related issue is the incorrect resolution of coreferences; each mistake leads to a misplaced argument. Most of these error sources can be eliminated, if the pre-processing improves. However, many difficulties simply arise from erroneous or ambiguous descriptions. Still, *fu*SE interprets most of them correctly. Most encouragingly, the average rank of the correct element is near 1. Thus, our scoring mechanism succeeds in placing the right elements on top of the list.

4 Evaluation

To measure the performance of *fu*SE on unseen data, we set up an end-to-end evaluation. We created two new scenarios. They take place in the kitchen setting again, but involve different actions and objects. In the first, subjects are supposed to teach the robot, how to start the dishwasher and in the second, how to prepare cereals. Once more we used *Prolific* to collect the data and set the number of participants to 110. However, we accepted only 101 submissions, i.e. 202 descriptions. We randomly drew 50 descriptions each. Since the evaluation of the overall approach entails the same output as the third stage, we prepared the gold standard like in subsection 3.4 and used the same ontologies. Table 11 details the dataset used in the end-to-end evaluation. Additionally, we provide five exemplary descriptions from the dataset in Table 14 (Appendix A).

In the end-to-end evaluation our approach synthesized 73 method signatures; five were missed due to an incorrect first-stage classification. Out of 73 synthesized methods we assessed seven to be inappropriate. Additionally, 36 parameters were mapped correctly and no false positives were created. Except for the missing method signatures the results are in line with the third-stage evaluation.

The results for the method body synthesis, as depicted in Table 10, even exceed the previous evaluation. The value of the F_1-score is 87.7% for

	pre.	rec.	F_1
methods	.924	.884	.904
parameters	.828	.951	.885
API calls	.735	.859	.792

Table 12: Evaluation results for the speech corpus.

individuals and 66.9% for entire API calls. Again, recall and F_1 increase, if we exclude SRL errors. However, the effect is smaller here. Moreover, the average rank is also closer to the optimum (1.0) in both cases. Since the first two stages of *fu*SE are based on neural networks, it is difficult to say why the results in the end-to-end evaluation improve. However, we believe the main cause is the introduction of a new test dataset, which has two consequences. First, the models used in the first two stages are learned on all four scenarios instead of three, i.e. the models are trained on a larger dataset, which (presumably) makes them more robust. Second, the new task may be simpler to describe. Consequently, the descriptions comprise simpler wordings and become easier to handle. In summary, the results show that *fu*SE generalizes to different settings – at least in the same domain – and is marginally degraded by error propagation.

To assess how well *fu*SE generalizes to truly spoken utterances we evaluated on another dataset. It is a collection of recordings from multiple recent projects. The setting (instructing a humanoid robot in a kitchen setting) is the same. However, none of the scenarios involved teaching new functionality. Thus, we can only measure *fu*SE's ability to construct scripts. The descriptions in this dataset comprise control structures to a much larger extent. Altogether the dataset comprises 234 recordings and manual transcriptions. The 108 subjects were mostly under-graduate and graduate students.

On the transcripts we assess the mapping of methods and parameters individually. The results for both and entire calls are depicted in Table 12. Even though the spoken samples comprise a vast number of disfluencies and grammatical flaws, *fu*SE maps more calls correctly. This counter-intuitive effect may be explained by the lower complexity and briefness of the spoken descriptions. Regarding the control structures, 27.4% were injected correctly. Note that *correctly* means an appropriate condition plus a block with correct extent. If we lower the standards for condition correctness, the share of correct structures is 71.23%.

5 Conclusion

We have presented *fu*SE, a system for programming in natural language. More precisely, we aim to enable laypersons to teach an intelligent system new functionality with nothing but spoken instructions. Our approach is three-tiered. First, we classify whether a natural language description entails an explicitly stated intent to teach new functionality. If an intent is spotted, we use a second classifier to separate the input into semantically disjoint parts; we identify declarative and specifying parts and filter out superfluous information. Finally, we synthesize method signatures from the declarative and method bodies from the specifying parts. Method bodies contain instructions and control structures. Instructions are mapped to API calls. We implemented the first two steps using classical machine learning and neural networks. Teaching intents are identified with an accuracy of 97.7% (using BERT). The classification of the semantics is correct in 97.6% of the cases (using a BiLSTM).

We evaluated *fu*SE on 100 descriptions obtained from a user study. The results are promising; *fu*SE correctly synthesized 84.6% of the method signatures. The mapping of instructions in the body to API calls achieved an F_1-score of 66.9%. In a second evaluation on a speech corpus the F_1-score for API calls is 79.2%.

We plan to evaluate *fu*SE in other domains. It will be interesting to see, if we can reuse (or transfer) the machine learning models as well as the rest of the approach. Future adoptions to *fu*SE will include the integration of a dialog component. We may query the user in case of ambiguous statements or missing parameters. We have implemented an extensible dialog module and shown that it can be used to resolve ambiguous references, word recognition errors, and missing conditions (Weigelt et al., 2018a). However, we still have to figure out, how to query users properly if an API mapping is ambiguous or parameters are missing. Another improvement concerns the analysis of verb references. Humans often refer to previous actions, which may cause superfluous instructions. We will also implement a sanity check that considers feasibility and meaningfulness of the sequence of actions in the method body. The latter may involve a feedback mechanism via the dialog component. Giving feedback to newly learned method definitions that may be lengthy and therefore unhandy to repeat as a whole is an interesting challenge.

References

Mattia Atzeni and Maurizio Atzori. 2018a. Towards Semantic Approaches for General-Purpose End-User Development. In *2018 Second IEEE International Conference on Robotic Computing (IRC)*, pages 369–376.

Mattia Atzeni and Maurizio Atzori. 2018b. Translating Natural Language to Code: An Unsupervised Ontology-Based Approach. In *2018 IEEE First International Conference on Artificial Intelligence and Knowledge Engineering (AIKE)*, pages 1–8.

Amos Azaria, Jayant Krishnamurthy, and Tom M. Mitchell. 2016. Instructable intelligent personal agent. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2681–2689, Phoenix, Arizona. AAAI Press.

Bruce W. Ballard and Alan W. Biermann. 1979. Programming in Natural Language: "NLC" As a Prototype. In *Proceedings of the 1979 Annual Conference (ACM)*, ACM '79, pages 228–237, New York, NY, USA. ACM.

Andrew Begel. 2004. Spoken Language Support for Software Development. In *Proceedings of the 2004 IEEE Symposium on Visual Languages - Human Centric Computing*, VLHCC '04, pages 271–272, USA. IEEE Computer Society.

Andrew Begel and Susan L. Graham. 2005. Spoken Programs. In *Proceedings of the 2005 IEEE Symposium on Visual Languages and Human-Centric Computing*, VLHCC '05, pages 99–106, USA. IEEE Computer Society.

Alan W. Biermann and Bruce W. Ballard. 1980. Toward Natural Language Computation. *Comput. Linguist.*, 6(2):71–86.

Alan W. Biermann, Bruce W. Ballard, and Anne H. Sigmon. 1983. An Experimental Study of Natural Language Programming. *International Journal of Man-Machine Studies*, 18(1):71–87.

Piotr Bojanowski, Edouard Grave, Armand Joulin, and Tomas Mikolov. 2017. Enriching word vectors with subword information. *Transactions of the Association for Computational Linguistics*, 5:135–146.

Giovanni Campagna, Rakesh Ramesh, Silei Xu, Michael Fischer, and Monica S. Lam. 2017. Almond: The Architecture of an Open, Crowdsourced, Privacy-Preserving, Programmable Virtual Assistant. In *Proceedings of the 26th International Conference on World Wide Web*, WWW '17, pages 341–350, Republic and Canton of Geneva, Switzerland. International World Wide Web Conferences Steering Committee.

Giovanni Campagna, Silei Xu, Mehrad Moradshahi, Richard Socher, and Monica S. Lam. 2019. Genie: A generator of natural language semantic parsers for virtual assistant commands. In *Proceedings of the 40th ACM SIGPLAN Conference on Programming Language Design and Implementation*, PLDI 2019, pages 394–410, Phoenix, AZ, USA. Association for Computing Machinery.

Xavier Carreras and Lluís Màrquez. 2004. Introduction to the CoNLL-2004 shared task: Semantic role labeling. In *Proceedings of the Eighth Conference on Computational Natural Language Learning (CoNLL-2004) at HLT-NAACL 2004*, pages 89–97, Boston, Massachusetts, USA. Association for Computational Linguistics.

Xavier Carreras and Lluís Màrquez. 2005. Introduction to the CoNLL-2005 shared task: Semantic role labeling. In *Proceedings of the Ninth Conference on Computational Natural Language Learning (CoNLL-2005)*, pages 152–164, Ann Arbor, Michigan. Association for Computational Linguistics.

Bo Chen, Le Sun, and Xianpei Han. 2018. Sequence-to-Action: End-to-End Semantic Graph Generation for Semantic Parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 766–777, Melbourne, Australia. Association for Computational Linguistics.

Ronan Collobert, Jason Weston, Léon Bottou, Michael Karlen, Koray Kavukcuoglu, and Pavel Kuksa. 2011. Natural Language Processing (Almost) from Scratch. *J. Mach. Learn. Res.*, 12:2493–2537.

Alain Désilets, David C. Fox, and Stuart Norton. 2006. VoiceCode: An Innovative Speech Interface for Programming-by-voice. In *CHI '06 Extended Abstracts on Human Factors in Computing Systems*, CHI EA '06, pages 239–242, New York, NY, USA. ACM.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. 2019. BERT: Pre-training of deep bidirectional transformers for language understanding. In *Proceedings of the 2019 Conference of the North American Chapter of the Association for Computational Linguistics: Human Language Technologies, Volume 1 (Long and Short Papers)*, pages 4171–4186, Minneapolis, Minnesota. Association for Computational Linguistics.

Li Dong and Mirella Lapata. 2018. Coarse-to-fine decoding for neural semantic parsing. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 731–742, Melbourne, Australia. Association for Computational Linguistics.

Christiane Fellbaum. 1998. *WordNet: An Electronic Lexical Database*. MIT Press.

Kelvin Guu, Panupong Pasupat, Evan Liu, and Percy Liang. 2017. From Language to Programs: Bridging Reinforcement Learning and Maximum Marginal Likelihood. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 1051–1062, Vancouver, Canada. Association for Computational Linguistics.

Armand Joulin, Edouard Grave, Piotr Bojanowski, and Tomas Mikolov. 2017. Bag of tricks for efficient text classification. In *Proceedings of the 15th Conference of the European Chapter of the Association for Computational Linguistics: Volume 2, Short Papers*, pages 427–431, Valencia, Spain. Association for Computational Linguistics.

Mathias Landhäußer, Sebastian Weigelt, and Walter F. Tichy. 2017. NLCI: A Natural Language Command Interpreter. *Automated Software Engineering*, 24(4):839–861.

Lauria Lauria, Guido Bugmann, Theocharis Kyriacou, Johan Bos, and Ewan Klein. 2001. Training personal robots using natural language instruction. *IEEE Intelligent Systems*, 16(5):38–45.

Stanislao Lauria, Guido Bugmann, Theocharis Kyriacou, and Ewan Klein. 2002. Mobile robot programming using natural language. *Robotics and Autonomous Systems*, 38(3):171–181.

Vu Le, Sumit Gulwani, and Zhendong Su. 2013. SmartSynth: Synthesizing smartphone automation scripts from natural language. In *Proceeding of the 11th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '13, pages 193–206, Taipei, Taiwan. Association for Computing Machinery.

Toby Jia-Jun Li, Igor Labutov, Xiaohan Nancy Li, Xiaoyi Zhang, Wenze Shi, Wanling Ding, Tom M. Mitchell, and Brad A. Myers. 2018. APPINITE: A Multi-Modal Interface for Specifying Data Descriptions in Programming by Demonstration Using Natural Language Instructions. In *2018 IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC)*, pages 105–114.

Nicholas K. Lincoln and Sandor M. Veres. 2012. Natural Language Programming of Complex Robotic BDI Agents. *Journal of Intelligent & Robotic Systems*, 71(2):211–230.

Hugo Liu and Henry Lieberman. 2005. Metafor: Visualizing Stories as Code. In *IUI '05: Proceedings of the 10th International Conference on Intelligent User Interfaces*, pages 305–307. ACM.

Mehdi Manshadi, Daniel Gildea, and James Allen. 2013. Integrating programming by example and natural language programming. In *Proceedings of the Twenty-Seventh AAAI Conference on Artificial Intelligence*, AAAI'13, pages 661–667, Bellevue, Washington. AAAI Press.

Irena Markievicz, Minija Tamosiunaite, Daiva Vitkute-Adzgauskiene, Jurgita Kapociute-Dzikiene, Rita Valteryte, and Tomas Krilavicius. 2017. Reading Comprehension of Natural Language Instructions by Robots. In *Beyond Databases, Architectures and Structures. Towards Efficient Solutions for Data Analysis and Knowledge Representation*, pages 288–301. Springer.

Hongyuan Mei, Mohit Bansal, and Matthew R. Walter. 2016. Listen, attend, and walk: Neural mapping of navigational instructions to action sequences. In *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI'16, pages 2772–2778, Phoenix, Arizona. AAAI Press.

Rada Mihalcea, Hugo Liu, and Henry Lieberman. 2006. NLP (Natural Language Processing) for NLP (Natural Language Programming). In *Proceedings of the 7th International Conference on Computational Linguistics and Intelligent Text Processing*, CICLing'06, pages 319–330, Berlin, Heidelberg. Springer-Verlag.

Tomas Mikolov, Edouard Grave, Piotr Bojanowski, Christian Puhrsch, and Armand Joulin. 2018. Advances in pre-training distributed word representations. In *Proceedings of the Eleventh International Conference on Language Resources and Evaluation (LREC-2018)*, Miyazaki, Japan. European Languages Resources Association (ELRA).

Andrea Moro, Alessandro Raganato, and Roberto Navigli. 2014. Entity linking meets word sense disambiguation: a unified approach. *Transactions of the Association for Computational Linguistics*, 2:231–244.

David Price, Ellen Riloff, Joseph Zachary, and Brandon Harvey. 2000. NaturalJava: A Natural Language Interface for Programming in Java. In *Proceedings of the 5th International Conference on Intelligent User Interfaces*, IUI '00, pages 207–211, New Orleans, Louisiana, USA. ACM.

Maxim Rabinovich, Mitchell Stern, and Dan Klein. 2017. Abstract Syntax Networks for Code Generation and Semantic Parsing. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics*, pages 1139–1149. Association for Computational Linguistics.

Lanbo She, Yu Cheng, Joyce Y. Chai, Yunyi Jia, Shaohua Yang, and Ning Xi. 2014. Teaching Robots New Actions through Natural Language Instructions. In *The 23rd IEEE International Symposium on Robot and Human Interactive Communication*, pages 868–873, Edinburgh, UK. IEEE.

Alane Suhr and Yoav Artzi. 2018. Situated Mapping of Sequential Instructions to Actions with Single-step Reward Observation. In *Proceedings of the 56th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 2072–2082, Melbourne, Australia. Association for Computational Linguistics.

Kristina Toutanova, Dan Klein, Christopher D. Manning, and Yoram Singer. 2003. Feature-rich part-of-speech tagging with a cyclic dependency network. In *Proceedings of the 2003 Human Language Technology Conference of the North American Chapter of the Association for Computational Linguistics*, pages 252–259.

Sida I. Wang, Samuel Ginn, Percy Liang, and Christopher D. Manning. 2017. Naturalizing a Programming Language via Interactive Learning. In *Proceedings of the 55th Annual Meeting of the Association for Computational Linguistics (Volume 1: Long Papers)*, pages 929–938, Vancouver, Canada. Association for Computational Linguistics.

Sebastian Weigelt, Tobias Hey, and Mathias Landhäußer. 2018a. Integrating a Dialog Component into a Framework for Spoken Language Understanding. In *Proceedings of the 6th International Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*, RAISE '18, pages 1–7, New York, NY, USA. ACM.

Sebastian Weigelt, Tobias Hey, and Vanessa Steurer. 2018b. Detection of Conditionals in Spoken Utterances. In *2018 IEEE 12th International Conference on Semantic Computing (ICSC)*, pages 85–92.

Sebastian Weigelt, Tobias Hey, and Vanessa Steurer. 2018c. Detection of Control Structures in Spoken Utterances. *International Journal of Semantic Computing*, 12(3):335–360.

Sebastian Weigelt, Tobias Hey, and Walter F. Tichy. 2017. Context Model Acquisition from Spoken Utterances. In *Proceedings of The 29th International Conference on Software Engineering & Knowledge Engineering*, pages 201–206, Pittsburgh, PA.

Sebastian Weigelt, Vanessa Steurer, Tobias Hey, and Walter F. Tichy. 2020. Roger that! Learning How Laypersons Teach New Functions to Intelligent Systems. In *2020 IEEE 14th International Conference on Semantic Computing (ICSC)*, pages 93–100.

William E. Winkler. 1990. *String Comparator Metrics and Enhanced Decision Rules in the Fellegi-Sunter Model of Record Linkage*.

Terry Winograd. 1972. Understanding natural language. *Cognitive Psychology*, 3(1):1–191.

A Dataset Examples

The dataset includes descriptions of varying quality. Some texts have syntactical flaws such as typos and grammar mistakes. They also vary in terms of descriptiveness and style; the latter ranges from full sentences to notes. Table 13 shows six examples from the preliminary study (scenarios one to four) and Table 14 five examples from the end-to-end evaluation (scenarios five and six). Most of the descriptions contain errors. For instance, description 2180 contains typos, such as "ring some beverage".

B Architectures and Hyper-parameters

We applied a broad range of machine learning approaches to the classification tasks. Table 15 shows the types, architectures and hyper-parameters we tested in the process. We also experimented with self-trained and pre-trained fastText embeddings.

C Configurations and Results

Table 16 shows representative configurations for the first stage of *fu*SE (binary classification); for neural networks we altered the hyper-parameters systematically to give an intuition of the effects. There are general trends. Classifiers perform better on randomly split data, a batch size of 100 is better than 300, and pre-trained embeddings outperform the self-trained in almost all cases. Overall, BERT-based classifiers achieve the best results. However, some neural network configurations come close (e.g. $\text{RNN}_{6.0}$); classical machine learning techniques are inadequate. For the second stage (ternary classification) we show interesting results in Table 17. The trends are as follows. The preferable batch size is 32, pre-trained embeddings again outperform the self-trained, and RNNs are best.

D Call Candidate Scoring

In subsection 3.4 we only discuss the rationale behind our call candidate scoring mechanism. Subsequently, we give a formal introduction. A call candidate is an API method with arguments (extracted from the natural language input). The arguments are of either primitive, composite (strings or enumerations), or previously defined types (e.g. objects from the environment). The arguments adhere to the formal definition of the API method. For each call candidate c *fu*SE calculates the score $S(c)$ as follows:

$$S(c) = \phi * P(c) * S_M(c) + (1-\phi) * WS_P(c) \quad (1)$$

The score is composed of two components: the method score $S_M(c)$ and the weighted parameter score $WS_P(c)$. The impact of the latter on the final score can be adjusted with the weight ϕ. Further, $S_M(c)$ is scaled by the perfect match bonus $P(c)$:

$$P(c) = \begin{cases} \tau & M(c) > 0.9 \\ 1 & \text{otherwise} \end{cases} \quad (2)$$

The perfect match bonus $P(c)$ allows us to prefer call candidates with a method name score $M(c)$ above 0.9. The scaling factor τ is configurable ($\tau \geq 1$). The method score $S_M(c)$ is computed as follows:

$$S_M(c) = M(c) - \frac{\beta}{|I_A(c)|} * \left(1 - \frac{|I_F(c)|}{|I_A(c)|}\right) \quad (3)$$

The method name score $M(c)$ is the maximal similarity of the natural language chunk that represents the action (or event) and the (API) method name. We use Jaro-Winkler and fuzzy score as similarity measures. To obtain the method score $S_M(c)$, the method name score $M(c)$ is reduced by a subtrahend that indicates how well the method name represents the words in the original natural language chunk. The subtrahend is composed of two factors. The second is one minus the fraction of words in the chunk that can be found in the method name and the total amount of words in the chunk; i.e., this factor is the share of unmapped words. The other factor scales it by a configurable parameter β, which is divided by length of the chunk. The rationale behind this is as follows. In short chunks each word is important. Therefore, unmapped words are strongly penalized. With an increasing number of words in the chunk, it is increasingly unlikely to map all words. However, in longer chunks many words are semantically irrelevant. Therefore, we reduce the subtrahend with the length of the chunk.

The weighted parameter score $WS_P(c)$ in Equation 1 is calculated as follows:

$$WS_P(c) = S_P(c) - \omega * Pen(c) \quad (4)$$

The score is composed of the parameter score $S_P(c)$ and a penalty value $Pen(c)$; the latter is weighted by the configurable factor ω. The parameter score $S_P(c)$ is calculated as follows:

$$S_P(c) = \sum P_i(c) * \frac{|P_M|}{|P_O(c)|} \quad (5)$$

ID	scen.	text
302	1	Look directly at the person. Wave your hand. Say 'hello'.
1000	2	You have to place the cup under the dispenser and press the red button to make coffee.
1346	2	Making coffee means you have to press the red button, put a cup underneath the hole and then pouring the coffee that comes out into your cup
2180	3	To ring a beverage, open the fridge and select one of te beverages inside, pour it into one of the glasses on the kitchen counter and hand the glass over to the person.
2511	4	collect cutlery from cupboard, bring them to the table and place down neatly
2577	4	To set the table for two, Go to the cupboard and take two of each; plates, glasses, knives, and forks. Take them to the kitchen table and set two individual places.

Table 13: Six exemplary submissions taken from the preliminary study dataset (scenarios one to four).

ID	scen.	text
E_10	5	Hey, Amar. We're going to start the dishwasher so what we have to do is first make sure the dishwasher is closed and then press the blue button twice to start the dishwasher.
E_79	5	Hi Armar. Turning on the Dishwasher means you have to go to the dishwasher. Close the dishwasher, then press the blue button 2 times.
E_81	5	Hi Armar, to use the dishwasher you need to check first if it is closed, if not, close it by pushing the front door. If it is closed, look for the blue button on the dishwasher. Once you find it, press it a first time and then a second time. That's how you start the dishwasher.
E_117	6	hi armar, you get your cereal ready you need to go to the fridge and open the door by pulling it. you will find the milk bottle inside the fridge door. lift it out and carry it to the kitchen table. place it next to your bowl and cereal box. start by filling the bowl with your cereal then pour in some milk.
E_158	6	Hi Armar, you have to go to the fridge, open it, grab the milk, close it and carry the milk to the kitchen table. Then place it next to the bowl and the cereal box. Fill the bowl with the cereals and then pour the mil in the bowl. That is how you prepare some cereals

Table 14: Five exemplary submissions taken from the end-to-end evaluation dataset (scenarios five and six).

P_M is the set of all parameters p_i (extracted from the natural language input) that were mapped to formal method parameters. Each p_i has a similarity score ($P_i(c)$). Thus, $S_P(c)$ is the sum of all similarity scores of mapped parameters multiplied with the share of mapped (P_M) and expected formal parameters as defined in the ontology ($P_O(c)$). To calculate $WS_P(c)$ (see Equation 4), $S_P(c)$ is reduced by the penalty value $Pen(c)$ that is calculated as follows:

$$Pen(c) = \frac{|P_E| - |P_M|}{|P_E|} \tag{6}$$

P_E is the set of parameters that were extracted from natural language input (see Figure 2). Thus, $Pen(c)$ is the number of parameters in the input that were not mapped to a formal method parameter, normalized by the total amount of extracted (natural language) parameters.

For the evaluation of the third stage of *fu*SE and the end-to-end-evaluation we set the method score weight ϕ to 0.6, the perfect match multiplier τ to 1.5, the search string coverage weight β to 0.5, and the penalty factor ω to 0.3. We determined all values empirically with the help of examples from scenario 4.

types	architect.	additional layers	number of units	epochs	batch sizes	dropout values	learning rates
ANN		Flatten (Flat), GMax, Dense (D), Dropout(DO)	10, 32, 50, 64, 100, 128, 200, 256	binary: 300, 500, 1000	binary: 100, 300	0.2, 0.3, 0.4	0.001, 0.0005
CNN	CONV	Max, GMax, Dense (D), Dropout(DO)		ternary: 50, 100, 300	ternary: 32, 64, 100, 300		
RNN	LSTM GRU BiLSTM BiGRU	Dense (D), Dropout (DO)					
BERT		Flatten (Flat)		5, 10, 300	32		0.00002

Table 15: Overview of the types, architectures, and hyper-parameters of neural networks used in the two classification tasks (step one and two of *fu*SE).

		batch	random		scenario	
name	additional layers	size	self-trained	fastText	self-trained	fastText
ANN$_{1.0}$	Flat, D(10)	100	(.907) .911	(.874) .887	(.918) .759	(.897) .722
ANN$_{2.0}$	Flat, D(100)	100	(.916) .914	(.846) .867	(.905) .781	(.874) .715
ANN$_{2.1}$	Flat, D(100)	300	(.921) .922	(.844) .870	(.922) .732	(.863) .577
ANN$_{3.0}$	GMax, D(10)	100	(.876) .887	(.872) .902	(.907) .766	(.905) .542
ANN$_{3.1}$	GMax, D(100)	100	(.899) .896	(.879) .896	(.893) .668	(.918) .674
ANN$_{3.2}$	GMax, D(100)	300	(.888) .889	(.877) .897	(.895) .676	(.908) .428
CNN$_{1.0}$	Conv(128, 3), GMax, D(10)	100	(.947) .966	(.954) .963	(.962) .765	(.966) .854
CNN$_{1.1}$	Conv(128, 5), GMax, D(10)	100	(.947) .971	(.930) .965	(**.973**) .743	(.973) .776
CNN$_{1.2}$	Conv(128, 7), GMax, D(10)	100	(.952) .966	(.943) .962	(**.973**) .775	(.970) .897
CNN$_{2.0}$	Conv(128, 3), Max(2), Conv(64, 3), GMax, D(10)	100	(.952) .959	(.952) **.971**	(.968) .855	(.962) .874
CNN$_{2.1}$	Conv(128, 5), Max(2), Conv(64, 5), GMax, D(10)	100	(.949) **.972**	(.952) .966	(.969) .850	(.975) .859
CNN$_{2.2}$	Conv(128, 5), Max(2), Conv(128, 5), GMax, D(10)	100	(.952) .964	(.954) .966	(**.973**) .862	(**.977**) .862
CNN$_{2.3}$	Conv(128, 5), Max(2), Conv(128, 5), GMax, D(10)	300	(.952) .953	(.947) .965	(**.973**) .783	(.971) .901
CNN$_{2.4}$	Conv(128, 5), Max(5), Conv(128, 5), GMax, D(10)	100	(**.956**) .958	(.952) .959	(.962) .901	(.973) .801
RNN$_{1.0}$	GRU(128)	100	(.560) .625	(.562) .625	(.477) .299	(.519) .702
RNN$_{1.1}$	GRU(128), D(100)	100	(.562) .625	(.562) .625	(.519) .702	(.519) .702
RNN$_{2.0}$	BiGRU(32), DO(0.2), D(64), DO(0.2)	100	(.947) .944	(.952) .959	(.954) .911	(.958) **.932**
RNN$_{3.0}$	LSTM(64)	100	(.566) .631	(.568) .638	(.519) .702	(.519) .702
RNN$_{3.1}$	LSTM(128)	100	(.570) .625	(.654) .738	(.519) .702	(.519) .702
RNN$_{4.0}$	LSTM(128), D(100)	100	(.562) .625	(.562) .625	(.519) .702	(.519) .702
RNN$_{4.1}$	LSTM(128), D(100)	300	(.562) .625	(.567) .633	(.519) .702	(.519) .702
RNN$_{5.0}$	BiLSTM(64), DO(0.2), D(64), DO(0.2)	100	(.947) .955	(.949) .955	(.956) .896	(.962) .916
RNN$_{5.1}$	BiLSTM(64), DO(0.2), D(64), DO(0.2)	300	(.929) .919	(.954) .949	(.945) .650	(.966) .872
RNN$_{5.2}$	BiLSTM(64), DO(0.3), D(200), D(100)	100	(.941) .947	(.947) .949	(.947) .884	(.956) .911
RNN$_{6.0}$	BiLSTM(128), D(64)	100	(.951) .955	(**.956**) .959	(.960) **.927**	(.962) .919
RNN$_{6.1}$	BiLSTM(128), D(64), D(32)	100	(.945) .962	(.947) .955	(.950) .919	(.966) .898
RNN$_{7.0}$	BiLSTM(128), D(100), DO(0.3), D(50)	100	(.936) .937	(.945) .941	(.937) .922	(.954) .917
RNN$_{7.1}$	BiLSTM(128), D(100), DO(0.3), D(50)	300	(.934) .934	(.938) .947	(.937) .704	(.950) .907
RNN$_{8.0}$	BiLSTM(256), D(128)	100	(.952) .944	(.945) .952	(.954) .843	(.962) .912
BERT$_1$	5 epochs	32	(.973) .981		(.991) .969	
BERT$_2$	10 epochs	32	(**.976**) **.982**		(**.992**) .973	
BERT$_3$	300 epochs	32	(.962) **.982**		(**.992**) **.977**	
Decision Tree			(.893) .903		(.861) **.719**	
Random Forest			(.917) .909		(**.893**) .374	
Support Vector Machine			(.848) .861		(.870) .426	
Naïve Bayes			(.771) .801		(.765) .300	
Logistic Regression			(**.927**) **.947**		(.891) **.719**	
baseline (ZeroR)			.573		.547	

Table 16: Classification accuracy obtained on the validation (in parenthesis) and the test set for the first stage (binary classification). The best results (per classifier category) are printed in bold type. The basic structure of each neural network includes an embedding layer and an output layer (dense layer).

		batch	random		scenario	
name	additional layers	size	self-trained	fastText	self-trained	fastText
$ANN_{1.0}$	-	32	(.851) .855	(.851) .856	(.850) .779	(.851) .826
$ANN_{1.1}$	-	100	(.851) .849	(.852) .849	(.849) .746	(.851) .826
$ANN_{2.0}$	D(10)	32	(.848) .857	(.852) .849	(.850) .825	(.851) .826
$ANN_{2.1}$	D(100)	32	(.853) .856	(.853) .848	(.851) .822	(.851) .827
$RNN_{1.0}$	LSTM(64)	32	(.977) .976	(.979) .978	(.971) .960	(.975) .966
$RNN_{1.1}$	LSTM(64)	100	(.973) .972	(.978) .975	(.969) .952	(.974) .964
$RNN_{1.2}$	LSTM(128)	32	(.974) .976	(.978) .977	(.973) .960	(.973) .964
$RNN_{1.3}$	LSTM(128)	100	(.974) .975	(.978) .977	(.970) .962	(.971) .965
$RNN_{1.4}$	LSTM(128)	300	(.973) .973	(.977) .974	(.968) .954	(.972) .961
$RNN_{2.0}$	LSTM(128), DO(0.2)	32	(.976) .977	(.977) .977	(.970) .960	(.973) .966
$RNN_{2.1}$	LSTM(128), DO(0.4)	32	(.976) .977	(.979) .979	(.971) .959	(.974) .967
$RNN_{3.0}$	LSTM(128), D(64)	32	(.973) .972	(.977) .976	(.970) .955	(.971) .963
$RNN_{4.0}$	BiLSTM(64)	32	**(.987)** .984	(.987) .985	(.982) .949	(.981) .972
$RNN_{4.1}$	BiLSTM(64)	100	(.981) .980	(.986) .984	(.979) .960	(**.982**) .967
$RNN_{4.2}$	BiLSTM(128)	32	(.986) .983	(.987) .985	(**.983**) .960	(.981) **.976**
$RNN_{4.3}$	BiLSTM(128)	64	(.984) .983	(.987) .984	(.979) .952	**(.982)** .973
$RNN_{4.4}$	BiLSTM(128)	100	(.985) .983	(.986) .984	**(.983)** .960	(.981) .969
$RNN_{4.5}$	BiLSTM(128)	300	(.983) .982	(.985) .984	(.977) .956	(.980) .968
$RNN_{5.0}$	BiLSTM(128), D(64)	32	(.980) .983	(.985) .984	(.973) .960	(.979) .965
$RNN_{6.0}$	BiLSTM(128), D(100), DO(0.3), D(50)	32	(.982) .982	(.982) .985	(.978) .955	(.981) .968
$RNN_{7.0}$	BiLSTM(128), DO(0.2)	32	(.985) .984	(**.988**) **.988**	(.982) .958	(.981) .975
$RNN_{7.1}$	BiLSTM(128), DO(0.4)	32	(.985) **.986**	(.986) .986	(.980) .961	(.980) .973
$RNN_{7.2}$	BiLSTM(256), DO(0.2)	32	(.986) .984	(.987) .985	(.982) **.964**	(**.982**) .975
$RNN_{8.0}$	BiGRU(128)	32	(.984) .984	(.985) .985	(.976) .955	(**.982**) .968
$BERT_1$	5 epochs	32	(.979) .982		(.979) .965	
$BERT_2$	10 epochs	32	(**.983**) **.985**		(.983) .972	
$BERT_3$	300 epochs	32	(.981) .983		(**.985**) **.973**	
baseline (ZeroR)			.759		.757	

Table 17: Classification accuracy obtained on the validation (in parenthesis) and the test set for the second stage (ternary classification). The best results are printed in bold type. The basic structure of each model includes an embedding layer and an output layer (dense layer).