

PREPRINT

A layered Pipeline for Natural Language Robot Programming with Control Structures

Sascha Sucker and Dominik Henrich

Chair for Robotics and Embedded Systems, University of Bayreuth,
Universitätsstraße 30, D-95447 Bayreuth, Germany,
{sascha.sucker|dominik.henrich}@uni-bayreuth.de,
<https://robotics.uni-bayreuth.de>

Abstract. Natural language is an intuitive interface to supplement programming in modern automation settings. However, most natural language frameworks are specialized and not universally applicable. We contribute a novel layered pipeline that transforms the instructions of laypersons into robot programs with non-linear control flow and that facilitates reuse. The instructions are analyzed regarding grammatical features. From this, the syntactical analysis derives programs with nested control structures and references to physical parts within a scene to be manipulated. These programs are semantically interpreted during online execution in a concrete scene – i.e., the control structures are evaluated, and part specifications are grounded to physical parts. With that, a fully specified skill is created and executed by a robot system. Since only the input/output interface of each pipeline stage is defined, they are adjustable independently of each other. Our experiments demonstrate how industrial robots in diverse domains can be verbally programmed using the pipeline.

Keywords: Dependency Grammar · Modular Architecture · Intelligent Robots · End-User Programming · Syntax · Semantics.

1 Introduction

Shorter innovation cycles and small-batch production pose significant challenges for automation [4]. One response is to increase the accessibility of robot programming [17, 18, 24]. We envision future programming to feel equally natural as instructing human co-workers. Inspired by the main human interaction method [23], we follow the approach of *natural language programming*. However, the few currently existing natural language programming frameworks in robotics are limited to specific applications and are thus not universally applicable.

We therefore aim to advance natural language programming of industrial robots (Figure 1). Compared to mere robot commanding our notion of programming differs in the utilization of control structures, thus allowing the control flow of the program to be adapted online. We distinguish between the typically used control structures: Sequences, selections, and loops. The latter can

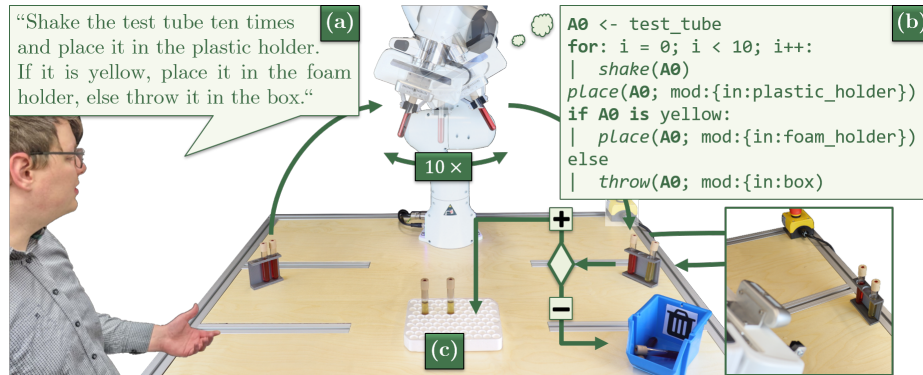


Fig. 1: Natural language (a) is transformed into programs with control structures (b). Using the program, the robot consecutively applies operations to the current scene (c).

be further subdivided into number-, condition-, and set-controlled loops. Other typical control structures are not explicitly considered, as they can be composed of the control structures above (e.g. switches or post-test loops). To facilitate a high degree of flexibility and thus quick adaptations to the desired domain, the approach must be modular. Thus, the programming system should be divided into independent sub-components connected with clearly defined input/output interfaces. This can be achieved using a pipeline architecture. The transformation of the spoken instruction into robot movements traverses different layers of abstraction – hence, resulting in a layered pipeline.

We present a layered pipeline for natural language robot programming (Figure 3). Our contribution is twofold: (i) We transfer the concepts of natural language programming using control structures to the context of industrial robots. (ii) Owing to the modular design, individual pipeline stages can be adjusted with little effort, facilitating reuse and quick adaptation to the desired domain.

2 Related Work

A common method of instructing a robot is to command pre-implemented robot skills. The language system interprets the verbal instruction by determining the skill type from the verb and deriving parameters from its valences – i.e., modifiers of the verb [6, 15, 21, 24]. However, this approach without additions does not allow the programming of complex programs since the control flow is not adapted. Nevertheless, we build upon this approach in order to define the operations embedded within the control structures (Section 3.1).

The morphology and grammatical dependencies of a naturally instructed sentence can be analyzed to derive control structures: For example, conjunctions of main clauses can be converted into sequences with the help of feasibility calculations [9]; selections are derived from the conditional member clauses [20]; condition- and amount-controlled loops can be generated from function words

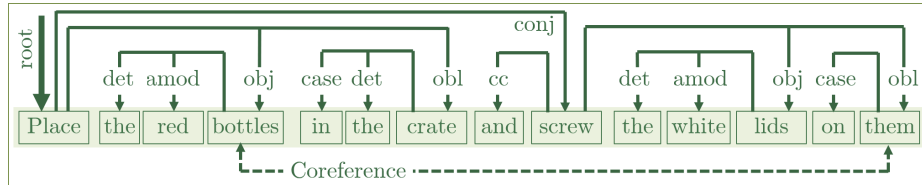


Fig. 2: The dependency tree encodes grammatical connections between words in a sentence. These dependencies are abbreviated with short keywords, which include determiners (det), adjective modifiers (amod), objects (obj), obliques (obl), and conjunctions (conj). Coreferences specify groups of words referring to the same thing.

like “while” or from mentioning the number of repetitions [8, 11]. Set-supervised loops can be programmed by defining the set of parts to be manipulated [5]. However, these approaches only focus on a few distinct control structures.

In contrast, analyzing the dependency tree of the instruction is a promising approach to derive all relevant control structures (as shown in [7, 22]). Dependency trees represent the syntactical make-up of an instruction by connecting words based on their grammatical relation (Figure 2) [10]. Starting from a root word, dependencies (e.g. subject, adjective, conjunction) are established to other sentence constituents. Through this, grammatical composition can be analyzed uniformly. Coreferences identify references between mentions of the same identity [3] – e.g. in Figure 2 ‘them’ refers to ‘bottles’.

In this paper, we expand on the work of Landhäußer et al. [7] and Weigelt et al. [22] in the context of flexible robot programming. So far, their approach has only been exemplarily applied to two specific robotic systems (humanoid/mobile robot). It neither includes a conceptual pipeline for natural language robot programming nor provides an adequate interface to cover diverse robot domains. Additionally, the grounding of underspecified part specifications to concrete parts (as discussed in our previous work [18]) is implemented prototypically only. This work addresses these problems to enable natural language programming for arbitrary robot systems utilizing dependency analysis.

3 Approach

The goal of our layered pipeline is to transform spoken instructions into robot movements which perform the desired manipulation of parts within the scene (Figure 3). This pipeline subdivides into abstraction and concretization stages. The former converts the spoken instructions into a program. Consequently, the abstraction includes Automatic Speech Recognition (ASR), Dependency Parsing, and Syntactical Analysis. We regard those stages as abstraction since information is primarily stripped from the instruction – ASR removes the explicit octave of the speech, and syntactical analysis may extract equal program statements from different sentence structures. The concretization stages interpret the program, thus transforming it into the desired robot motions for a concrete

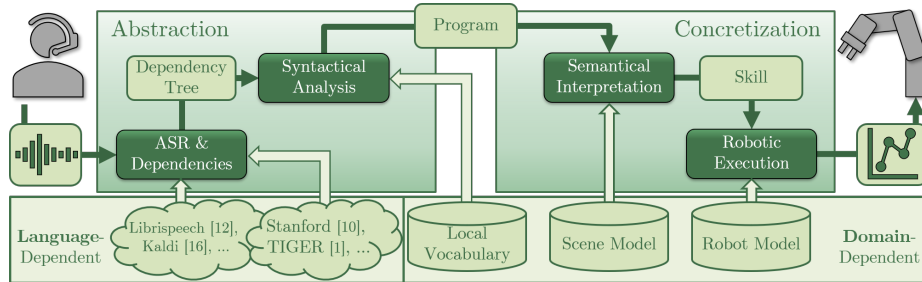


Fig. 3: The layered pipeline transforms spoken commands into fitting robot movements: Such commands are converted to text, whose grammatical dependencies are analyzed. These are parsed into a program with control structures. The program is interpreted for a scene to create skills, which are executed by the robotic system.

scene. To further increase modularity, the concretization has two stages: During Semantical Interpretation, the program is gradually interpreted online yielding robot skills, which the Robot Execution stage converts into concrete trajectories. Hence, the program is transformed to increasingly more explicit representations.

In more detail, the instructions are transformed by this pipeline as follows: The ASR stage transcribes the spoken audio signal into text. This text then is parsed with regard to its grammatical dependencies (Section 2). Program code is synthesized by exploring these dependencies and matching keywords to the local vocabulary during the Syntactical Analysis (Section 3.1). The instruction provides an implicit execution flow that must be transformed into concrete control structures with associated robot operations and part specifications. The resulting program is scene-independent, allowing it to be used with different task variants – e.g. by not having to precisely define the position of the individual parts in advance. Accordingly, the program is executed by providing a scene state during the Semantical Interpretation (Section 3.2), wherein the underspecified part specifications and conditionals are grounded to physical parts. The specific robot skill is then determined based on the grounding and the previous program state. Utilizing the skill definition of Pedersen et al. [14], hardware independence is guaranteed during this stage. Finally, the skills are transformed into concrete motion sequences that achieve the desired manipulation of the scene.

Both ASR and Dependency Parsing are studied extensively within computational linguistics. Thus, optimized methods are readily available. Robot Execution mostly follows the approach of executing skills [14]. In this paper, we therefore mainly cover the Syntactical Analysis and Semantical Interpretation.

3.1 Syntactical Analysis

The Syntactical Analysis converts dependency trees into programs (Figure 4a). We require such programs to represent the instructed parts, operations, and control structures. Based on this, the program is divided into declaration and procedure sections. The *declaration* lists all named part specifications \hat{P} and

PREPRINT

“Place three blue bottles in the crate and screw the white lids on them.”	Depend.	Mark	Control Structure
----- Declaration ----- A0 <- bottle(num: 3, col: blue) A1 <- lid (num: *, col: white) ----- Procedure ----- place(A0; mod: {1:field, to:right}) screw(A1; mod: {on:A0})	conj	And	Sequence
	sconj	If	Selection – Positive
	parataxis	Instead	Selection – Negative
	sconj	While	Conditional-loop
	sconj	Until	Conditional-loop (Neg.)
	obl	Times	Count-loop

(a) Example program.

(b) Excerpt of the control structure table.

Fig. 4: Programs are synthesized from instructions utilizing their grammatical dependencies (a). The control structures connecting statements are derived from a table of grammatical dependencies and function words (b).

assigns unique labels to each of them. We call each declaration list entry a variable, distinguishing between atomics and compounds. *Atomic variables* always refer to one part specification, whereas *compounds* link several atomic or other compound variables with one common operator ('and', 'or'). Depending on the domain, part specifications $\tilde{p} \in \tilde{P}$ can include features such as the type p_t , number p_n , color p_c , or location p_l (see Section 3.2) – resulting in the tuple

$$\tilde{p} = (p_n, p_t, p_c, p_l). \tag{1}$$

The *procedure section* contains the control flow and operations that access the declared part variables. We group the operations and the control structures into the superordinate term *statement*. Each operation is defined by its type, part variable to be manipulated, and other instructed modifications that describe the process in more detail. Programs are created by analyzing the dependency tree. Starting from the root node, the dependencies linking nodes within the tree are transformed into program constituents based on their mark (e.g. 'if' within a subordinated conjunction). The operation is identified by the verb node and its dependencies [21]. These include the parts to be moved (accusative object) and other specifying parameters (e.g. adverbs or obliques). A part specification is created if a dependency relates to a nominal node whose identifier is assigned to a part type in the language model. By analyzing its connected dependencies, additional part features are extracted (e.g. adjectives, numerals, or compounds).

A table of dependencies contains the control structures between statements with associated function word markings (Figure 4b). This method is reasonable since only few distinct function words exist (e.g. 'if', 'while', 'until') and they are rarely altered [2] – resulting in a table with few entries. Several statements are linked to a sequence employing chained conjunctions of main sentences with the marker 'and' (coordinating conjunction dependency). If the user instructs several sentences, the determined partial programs are likewise strung together as one sequence. However, humans rarely structure instructions concisely without ambiguities or contradictions so that the desired instruction sequence does not necessarily correspond to the one spoken [13]. Therefore, the feasibility of the possible orderings must be considered [9]. Selections are identified by the de-

pendency subordinated conjunction (sconj) and the associated marker 'if'. The condition is derived from the subordinate sentence and its positive case from the main sentence. If there is additionally a parataxis dependency on the main clause (e.g. with an 'instead' marker), the subordinate clause is considered the negative case. Conditional loops are determined analogously to the selections using e.g. the marking 'while', whereby the main clause corresponds to the contents of the loop body. Repetitions of an operation (e.g. "Press it five times") can be determined using an oblique dependency resulting in an amount loop. Set-loops are not derived directly from this table since they are inherently encoded in the part specifications by defining the required number of parts (e.g. "Move four cubes"). The operation performs similarly for all specified parts resembling a set-loop.

Occasionally, a part should be manipulated multiple times in succession. This can be taken into account by analyzing the coreferences as follows: Only references between nominals related to parts are considered, though one compound may also combine multiple references to such nominals. We obtain this set of part-related nominals \mathcal{N} by analyzing the dependency tree and the local vocabulary, where each id refers to the word at the corresponding position in the instruction. We use the set of coreference chains \mathcal{K} , where one coreference chain $k = (k_{\text{msm}}, k_{\text{ref}})$ contains two word id sets: k_{msm} (the most specific mentions) and k_{ref} (all references to k_{msm}). We define the word id set of all atomic mentions

$$\mathcal{A} = \{w \in \mathcal{N} \mid (\forall k \in \mathcal{K} : w \notin k.k_{\text{ref}})\} \quad (2)$$

as a subset of \mathcal{N} , whose ids are not referenced in any coreference ($k.k_{\text{ref}}$). Every atomic mention is transformed into an atomic variable with corresponding part specification (Equation 1). Compounds link atomics or other compounds via one conjunction. Accordingly, $\forall k_c \in \mathcal{K}_c \subseteq \mathcal{K}$ with

$$\mathcal{K}_c = \{k \in \mathcal{K} \mid |k.k_{\text{msm}}| > 1\} \quad (3)$$

one compound must be formed by linking the appropriate atomics or compounds of the $k.k_{\text{msm}}$. The remaining conjunctions of atomics and compounds without coreferences should also be usable as contiguous variable in the operation. For example, a sentence 'Move the bottle and lid to the left' should result in two atomic ('bottle', 'lid') and one compound variables. Therefore, such conjunctions must also be linked within a compound variable. Following the outlined stages, we can transform natural instructions into programs with nested control flow.

3.2 Semantical Interpretation

In this stage, the program is interpreted within a scene resulting in robot skills with concrete parameters. For this purpose, the program is processed sequentially, and the part specifications in the current program stage are grounded (Figure 5). Operations are converted to a concrete skills utilizing grounding as introduced in our previous work [18]. Grounding refers to assigning physical parts to part specifications that are associated to atomic variables. If a grounding was already found for one variable and is referenced, the existing grounding result is

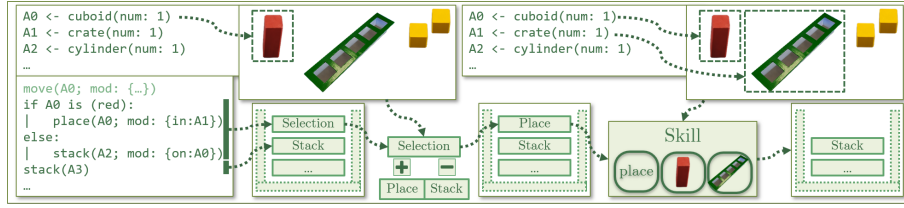


Fig. 5: The program is interpreted and consecutively transformed into precise skills, partially grounding the part specification on demand. Here, the first operation (move) was already executed, and the grounding for 'A0' is reused to interpret the selection.

reused, allowing repeated manipulations. The parts are grounded dynamically during the execution – i.e. only a subset of parts in the scene are grounded at one execution stage. This is for two reasons: (i) The scene state and, thus, future assignments may change; and (ii) the parts specified in the body of a selection must only be present if its condition resolves.

The operation is converted to a concrete skill utilizing grounding [18]. The current world state $\hat{P} = \{\hat{p}_0, \hat{p}_1, \dots\}$ is the set of the part states in a given scene. One state encompasses the relevant and identified features of the part (e.g. geometrical shape, color, or position). In contrast, part specifications \tilde{P} define boundary conditions that part states must satisfy in the context of an operation execution. To use the concept from [18] we convert every specification into p_n equal part templates with the same properties. Such part templates $P = \{p_0, p_1, \dots\}$ are defined equal to the specifications without the amount p_n . Therefore, we utilize the function $\gamma : P \rightarrow \hat{P}$, which maps all templates injectively to a part state, where each mapping must suffice the function $\sigma : \hat{P} \times P \rightarrow \{\text{TRUE}, \text{FALSE}\}$. Function σ return whether a part state \hat{p} satisfies the boundaries defined by part template p . Thus, for each part template p , exactly one state \hat{p} must be found that satisfies the boundaries of p . This general construction allows grounding to be performed on arbitrary domains. Since a local grounding is performed in this case, the states can be assigned to the templates in a greedy manner. Grounding allows skills to be filled with concrete parameters.

For example, a template $p = (p_t, p_c, p_l)$ might contain information about the geometric type p_t , the color p_c , and location p_l (based on Equation 1). A corresponding state $\hat{p} = (\hat{p}_t, \hat{p}_c, \hat{p}_l)$ is analogous in structure. Hereby, named features (such as type or color) can be described as an entry within a taxonomy. Such a taxonomy captures “is-a”-relations between a set of nodes T . Leaf nodes $\hat{T} \subset T$ denominate *concrete features* which part state may exhibit. When ascending from leaf nodes upwards towards the root node, encountered inner nodes encode increasingly abstract descriptions. Thus, such inner nodes may occur exclusively in similarly abstract part templates. An example of a taxonomy of part types in the palletizing domain might include the term 'box', which covers products ranging from small tea boxes to large packages. The set of part locations $L = L_s \cup L_u$ includes well-specified affine transformations (L_s) and underspecified constraints on the part transformation (L_u). Hence, an underspecified location e.g. describes

an area in scene space in which a part should be present. Each location $l \in L$ is associated with a *location function* $\text{is_at} : \hat{p} \times l \rightarrow \{\text{TRUE}, \text{FALSE}\}$, which outputs whether the pose of \hat{p} matches the location l . Analogously to named features, part states may only exhibit locations $l_s \in L_s$, while templates underlie no such restriction. In this domain, the satisfies function σ would correspond to

$$\sigma(\hat{p}, p) = \text{is_a}^{\text{type}}(\hat{p}_t, p_t) \wedge \text{is_a}^{\text{color}}(\hat{p}_c, p_c) \wedge \text{is_at}(\hat{p}, p_l). \quad (4)$$

The program is executed utilizing a call stack analogous to programming languages such as C++. Statements are pulled successively, where operations can be transformed into skills and control structures added to the call stack depending on their type. Sequences put their statements in reverse order, making the following pulled statement equal to the first one. Selections push either the positive or negative case based on their evaluation. If the head of a loop resolves, the loop is pushed again along with the body statement. Thereby, the body is pulled next, and the loop head is rechecked afterward. The resulting grounded skills can be executed on any suitable robot system.

4 Experimental Validation

We designed four benchmark tasks to highlight specific aspects of natural language programming using our proposed architecture (Figure 6): In *Task T1*, 'labeled' is an additional binary feature of part states corresponding to a taxonomy with an agnostic root node and two leaf nodes. The subordinated conjunction with 'if'-marking is transformed into a selection with the 'labeled' condition and positive/negative branches. This task shows that the definition of parts can be adapted by adding or removing features. With *Task T2*, an electrical connection is completed within an assembly benchmark toolkit [19]. One arbitrary conductor and one resistance conductor must be inserted in a pre-assembled electrical circuit. The general concept of our part types allows part states with unique naming. Therefore, the goal locations of the conductors can be uniquely identified by directly addressing the plates (e.g. 'P0' and 'Q0'), bypassing the possibly ambiguous grounding of part states. However, this task also highlights the drawback of local grounding: Based on the type taxonomy, 'resistance' is the child of the 'conductor' type. For example, the resistance may be mistakenly grounded for the first operation, leaving no matching part for the second operation. *Task T3* shows a count-loop within a laboratory domain. The content of a measuring cylinder is shaken five times, resulting in a for-loop with the upper bound parametrization of five. Due to the general skill definition, skills more complex than pick-and-place may be defined (e.g. 'shaking' and 'emptying' the measuring cylinder). This relates both to the required robot movements and the applied manipulation to the parts (e.g., a 'mixed'-feature may be increased during shaking). *Task T4* displays a condition-loop in a service domain by instructing the cleaning of a whiteboard. The position-controlled skills in *T1-T3* can be extended with force-control. Each task is situated within its unique domain and utilizes different control structures – thus, demonstrating the flexibility and adaptability of our approach.

PREPRINT


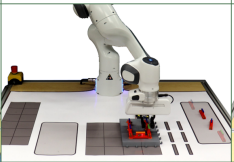


<i>T1</i> : “If 4 packages are labeled, place them on the palette, else lay them in the blue box.”	<i>T2</i> : “Place one conductor between P0 and Q0. Place the resistance between P1 and Q1.”	<i>T3</i> : “Shake the measuring cylinder five times and empty it into the bowl.”	<i>T4</i> : “While the whiteboard is not clean, swipe it with a sponge.”
<pre>A0 <- package(...) A1 <- palette(...) A2 <- box(...) ----- if A0 is labeled: place(A0; mod:{in:A1}) else: lay (A0; mod:{in:A2})</pre>	<pre>A0 <- conductor(...) A1 <- P0(...); A2 <- Q0(...) C0 <- ([A1, A2], and) A3 <- resistance(...) A4 <- P1(...); A5 <- Q1(...) C1 <- ([A4, A5], and) ----- place(A0; mod:{between:C0}) place(A3; mod:{between:C1})</pre>	<pre>A0 <- measuring_cylinder(...) A1 <- bowl(...) ----- for: i = 0; i < 5; i += 1: shake(A0) empty(A0; mod:{into:A1})</pre>	<pre>A0 <- whiteboard(...) A1 <- sponge(...) ----- while A0 is not:clean: swipe(A0; mod:{with:A1})</pre>
			

Fig. 6: Four benchmark tasks highlight the system’s capabilities. The first row shows the instruction(s), which are transformed into programs (second row). The third row displays a robot executing the instruction in the context of a corresponding domain.

5 Conclusion and Future Work

In this paper, we contributed a modular pipeline for natural language robot programming by laypersons. We achieved this by analyzing the grammatical speech patterns within the implicit instructions and transforming them into programs with explicit control structures (Section 3.1). Thus, we synthesized operations and part specifications embedded within control structures from language. By grounding these specifications, we can parametrize the operations to be suitable for industrial robots (Section 3.2). We showed the high adaptability of our pipeline by customizing it for diverse domains (Section 4).

Our approach may be extended in future work: (i) Currently, interpretation errors are propagated through the pipeline without mechanisms to seek user feedback. A dialog component may resolve this error propagation. (ii) Additionally, we defined the input/output within the pipeline to be human-readable. Thus, intermediate results may help by error detection and correction. (iii) The local grounding may lead to additional errors (Section 4) and must therefore be extended to a global grounding approach [18]. (iv) Furthermore, behavioral programming can be incorporated into natural language such that the programming more closely resembles the training of human co-workers. (v) Finally, we plan to compare the usability of our prototype with other programming systems.

Acknowledgement

This work was partly funded by the Deutsche Forschungsgemeinschaft (DFG) under grant agreements He2696/18 (VerbBot). The authors would like to thank Carsten Scholle (B.Sc.), Philipp Jahn (B.Sc.), and Siegfried Köhler (M.Sc.) for their valuable assistance in implementing and testing the demonstrator.

References

1. Brants S et al. (2004) TIGER: Linguistic Interpretation of a German Corpus. *Journal of Language and Computation*.
2. Carnap R (1937) *The Logical Syntax of Language*. Routledge & Kegan Paul.
3. Clark K, Manning CD (2016) Deep reinforcement learning for mention-ranking coreference models. *Association for Computational Linguistics: EMNLP*.
4. Dietz T et al. (2012) Programming System for Efficient Use of Industrial Robots for Deburring in SME Environments. *German Conf. on Robotics*. Munich, DEU.
5. Han J et al. (2020) Structuring Human-Robot Interactions via Interaction Conventions. *IEEE Int. Conf. on Robot and Human Interactive Comm. (RO-MAN)*.
6. Knoll A et al. (1997) Instructing cooperating assembly robots through situated dialogues in natural language. *IEEE Int. Conf. on Robotics and Automation (ICRA)*.
7. Landhäußer M, Hug R (2015) Text Understanding for Programming in Natural Language: Control Structures. *IEEE/ACM Int. Workshop on Realizing Artificial Intelligence Synergies in Software Engineering*. Florence, ITA.
8. Lauria S et al. (2002) Mobile robot programming using natural language. *Robotics and Autonomous Systems*.
9. Liu R, Zhang X (2018) Generating machine-executable plans from end-user's natural-language Instructions. *Knowledge-Based Systems*.
10. De Marneffe MC et al. (2014) Universal Stanford dependencies: A cross-linguistic typology. *Int. Conf. on Language Resources and Evaluation*. Reykjavik, ISL.
11. Matuszek C et al. (2013) Learning to parse natural language commands to a robot control system. *Experimental robotics*. Springer Heidelberg, DEU.
12. Panayotov V et al. (2015) Librispeech: an asr corpus based on public domain audio books. *2015 IEEE Int. Conf. on Acoustics, Speech and Signal Processing (ICASSP)*.
13. Pane JF et al. (2001) Studying the language and structure in non-programmers' solutions to programming problems. *Int. Journal of Human-Computer Studies*.
14. Pedersen M et al. (2016) Robot skills for manufacturing: From concept to industrial deployment. *Robotics and Computer-Integrated Manufacturing*.
15. Pires JN (2004) Robot-by-voice: Experiments on commanding an industrial robot using the human voice. *Industrial Robot: An International Journal*.
16. Povey D et al. (2011) The Kaldi Speech Recognition Toolkit. *IEEE 2011 Workshop on Automatic Speech Recognition and Understanding*.
17. Riedelbauch D, Hartwig J, Henrich D (2019) Enabling End-Users to Deploy Flexible Human-Robot Teams to Factories of the Future. *IROS 2019 Workshop*.
18. Riedelbauch D, Sucker S (2022) Visual Programming of Robot Tasks with Product and Process Variety. *Annals of Scientific Society for Assembly, Handling and Industrial Robotics* (to appear).
19. Riedelbauch D, Hümmer J (2022) A Benchmark Toolkit for Collaborative Human-Robot Interaction. *IEEE Int. Conf. on Robot and Human Inter. Comm. (RO-MAN)*.
20. Rybski P et al. (2007) Interactive robot task training through dialog and demonstration. *ACM/IEEE Int. Conference on Human-robot interaction*. New York, USA.
21. Spangenberg M, Henrich D (2015) Grounding of actions based on verbalized physical effects and manipulation primitives. *IEEE/RSJ IROS*.
22. Weigelt S, Hey T, Steurer V (2018) Detection of Conditionals in Spoken Utterances. *IEEE Int. Conference on Semantic Computing (ICSC)*. Laguna Hills, USA.
23. White L (1940) *THE SYMBOL: The Origin and Basis of Human Behavior*. ETC: A Review of General Semantics.
24. Wölfel K, Henrich D (2018) Grounding Verbs for Tool-Dependent, Sensor-Based Robot Tasks. *IEEE Int. Symp. on Robot and Human Inter. Comm. (RO-MAN)*.