Preprints of the 19th World Congress
The International Federation of Automatic Control
Cape Town, South Africa. August 24-29, 2014

Describing constraint-based assembly tasks in unstructured natural language [⋆]

Maj Stenmark and Jacek Malec

Dept of Computer Science, Lund University, 221 00 Lund, Sweden (email: [maj.stenmark,jacek.malec]@cs.lth.se)

Abstract: Task-level industrial robot programming is a mundane, error-prone activity requiring expertise and skill. Since humans easily communicate with natural language (NL), it may be attractive to use speech or text as instruction means for robots. However, there has to be a substantial amount of knowledge in the system to translate the high-level language instructions to executable robot programs.
In this paper, the method of Stenmark and Nugues (2013) for natural language programming of robotized assembly tasks is extended. The core idea of the method is to use a generic semantic parser to produce a set of predicate-argument structures from the input sentences. The algorithm presented here facilitates extraction of more complicated, advanced task instructions involving cardinalities, conditionals, parallelism and constraint-bounded programs, besides plain sequences of commands.
The bottleneck of this approach is the availability of easily parametrizable robotic skills and functionalities in the system, rather than the natural language understanding by itself.

Keywords: Robot programming, natural language, assembly, knowledge-based engineering

1. INTRODUCTION

Programming of a traditional robot cell requires considerable expertise and effort. The new generation of robots, that work in an unstructured environment, that might have more degrees of freedom and two arms, introduces an increased level of complexity in user interaction and instruction. Therefore, methods of robot instruction that are accessible to non-experts would lead to greater usability of industrial robotics. Yet another aspect of the problem lies in vendor-specific solutions, available for each brand of robots. Different tools of varying complexity, different robot programming languages and different abstraction levels of task descriptions make them inaccessible for a plain user.

Since humans communicate with natural language (NL), it may be attractive to use speech or text as instruction means for robots. This is non-trivial for two main reasons: First, NL is often ambiguous and its expressivity is richer than that of a typical programming language. Secondly, tasks can be expressed as goals as well as imperative statements, hence, even if the instructions are correctly interpreted, the description itself is often not enough to create a successful execution. There has to be a substantial amount of knowledge in the system to translate the high-level language instructions to executable robot programs.

In this paper, the simple method from Stenmark and Nugues (2013) for natural language programming of assembly tasks is extended. The core idea of the method is to use a generic semantic parser to produce a set of predicate-argument structures from the input sentences. The original algorithm allows extraction of only plain sequences of commands. Here we show that using the predicate-argument structures together with the dependency graphs facilitates also extraction of more complicated task instructions, which involve cardinalities (e.g., pick *two* bolts and *two* nuts), conditionals (e.g., if...then...else) and constraint-characterized programs (e.g., do...until...)

2. RELATED WORK

By abstracting away the underlying details of the system, e.g., by demonstration, high-level programming can make robot instruction accessible to non-expert users and reduce the workload for an experienced programmer. A survey of programming-by-demonstration models in robotics is presented by Billard et al. (2008).

In industrial robotics, programming and demonstration techniques are normally used to record trajectories and positions. As it is desirable to minimize downtime for the robot, much programming and simulation is done offline whereas only the fine tuning is done online (Hägele et al., 2008). There is a plethora of tools, often visual, for robot programming. In robotics, standardized graphical programming languages include Ladder Diagrams, Function Block Diagrams and Sequential Function Charts (IEC, 2003). Using a touch screen as an input device, icon-based programming languages such as in Bischoff et al. (2002) can also lower the threshold to robot programming.

Natural language programming for robots has been investigated since the early 1970s. SHRLDU (Winograd, 1971) is an example of the first attempts to give robots conversational competences. To interpret and convert a users sentences into instructions, robotic system often make

⋆ The research leading to these results has received partial funding from the European Union's seventh framework program (FP7/2007-2013) under grant agreements No. 285380 (project PRACE) and No. 287787 (project SMErobotics).

use of an intermediate representation. Examples include MacMahon et al. (2006) and Tellex et al. (2011), where the authors have developed their own domain-specific semantic representations for robot navigation.

Tenorth et al. (2010) parse pancake recipes in English from the World Wide Web and generate programs for their household robots. They use the WordNet lexical database (WordNet, 2010) with a constituent parser and they map entries in the WordNet dictionary to concepts in the Cyc ontology (Matuszek et al., 2006). Finally, they add mappings to common household objects.

In order to bridge the sentence to robot actions, all the examples above use ad-hoc formalisms. FrameNet (Ruppenhofer et al., 2010), based on frame semantics, is a comprehensive dictionary that provides a list of lexical models of the conceptual structures. Propbank (Palmer et al., 2005) has developed a extensive database of predicate-argument structures for verbs and nouns, and annotated large volumes of text. The Propbank nomenclature is used by most current statistical parsers, including ours.

Only few robotics systems use existing predicate-argument nomenclatures. An exception is RoboFrameNet (Thomas and Jenkins, 2012). However, the authors wrote their own frames inspired by FrameNet. They built a semantic parser that consists of a dependency parser and rules to map the grammatical functions to the arguments. Such techniques are known to have a limited coverage.

In the project described below we have used a multilingual high-performance statistical semantic parser (Björkelund et al., 2009, 2010) using the Propbank and Nombank lexicons. In contrast to RoboFrameNet, the parser we adopted can accept any kind of sentence. The NL processing module is a knowledge-based service in a larger programming environment (Stenmark and Malec, 2013). In particular, it allows one to create constraint-based task descriptions based on the iTaSC formalism, a property exploited here.

3. BACKGROUND

The system has been described in detail in our previous work (Stenmark and Nugues, 2013; Stenmark and Malec, 2013); a simplified view of its components is shown in Fig. 1. It is a cloud-based system for knowledge sharing and distributed AI reasoning. The knowledge and reasoning services are stored on a server called Knowledge Integration Framework (KIF), which contains data repositories and ontologies modeling objects and actions. KIF also provides servlets for planning, scheduling and code generation, as well as the NL-programming servlet described in this paper. These services are used for offline programming by the Engineering System (ES), which is a user-interface implemented as a plug-in to ABB RobotStudio (ABB Robotics, 2013) visual IDE.

Objects in the World The core ontology, **rosetta.owl** (Stenmark and Malec, 2013), contains devices such as sensors and robots. The ES also uses a separate ontology to describe parts, such as trays and workpieces. The ontologies describe object types and properties, while the data repositories contain instances of the types. E.g., a **ForceSensor** is a subtype of **Sensor** and of **PhysicalObject**, has property **measures** with value **Force**, and it also

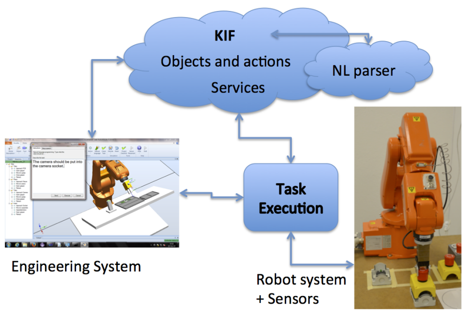
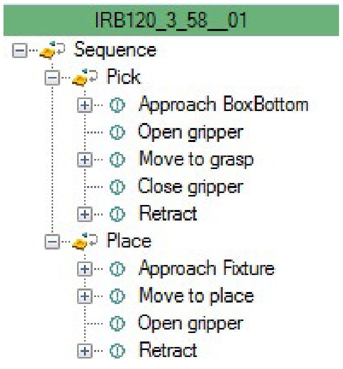


Fig. 1. A view of the system architecture.

inherits properties such as **weight** from **PhysicalObject**. The object types and their property types are later used by the natural language programming system to link arguments to real world objects. Objects are displayed by ES using their CAD models. Each object has a number of relative coordinate frames called *feature frames*, attached to its main object frame. The feature frames are used to express relations between objects. A typical case is a gripping pose described as a relation between a gripper frame and an object feature frame.

Task Vocabulary The task vocabulary is limited to existing robot capabilities. In the KIF repositories, robot actions are stored as program templates, called *skills*. There are primitive actions, such as *search*, *locate* and *move* which can be combined into more complex skills such as *pick* and *place*. Each skill has parameters, e.g., velocities, other objects, their feature frames, or relations. Each skill has also a set of device requirements, pre - and post-conditions as well as optional properties such as natural language labels. The skills are downloaded from the KIF libraries into the ES and added to a task sequence, see Fig. 2. This sequence can be edited by



IRB120_3_58__01
- Sequence
  - Pick
    - Approach BoxBottom
    - Open gripper
    - Move to grasp
    - Close gripper
    - Retract
  - Place
    - Approach Fixture
    - Move to place
    - Open gripper
    - Retract

Fig. 2. A sequence of skills.

drag-and-dropping objects and by editing parameters of each action or skill. As an additional modality, we have extended the system with natural language support for sequence generation. Using language to express a task is faster than downloading or selecting each skill separately; besides, speech allows hands-free instruction of the robot.

Natural Language Programming The task is expressed in unstructured English, either by typing it in a text box directly in the user interface, or by connecting an Android app to the ES and using its speech-to-text conversion. The text is sent to a servlet on KIF, which in turn calls a general purpose statistical parser[1] (Björkelund et al., 2010) that outputs predicate-argument structures in standard format (cf. Fig. 3).

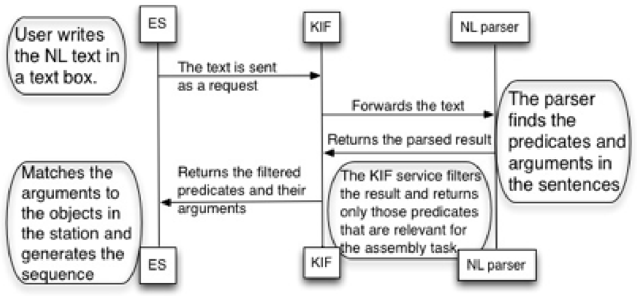

ES
KIF
NL parser
User writes the NL text in a text box.
The text is sent as a request
Forwards the text
Returns the parsed result
The parser finds the predicates and arguments in the sentences
Matches the arguments to the objects in the station and generates the sequence
Returns the filtered predicates and their arguments
The KIF service filters the result and returns only those predicates that are relevant for the assembly task
ES
KIF
NL parser

Fig. 3. The NL parsing sequence.

Predicate-Argument (PA) Structures As an example we use an assembly where a printed circuit board, a *PCB*, is covered with a metal plate, a *shieldcan*. First the PCB should be fixated, which can be expressed in English as *Take the PCB from the input tray and place it on the fixture.* The PA structures are *take(PCB, input tray)* and *place(it, fixture)*. The parser labels verbs with different *senses* depending on the context in which they are used. For example, take off (like a plane) is *take.19* and take down is *take.22*.

The parsing pipeline uses logistic regression to produce the PA structures, see Fig. 4. First, the dependency graph

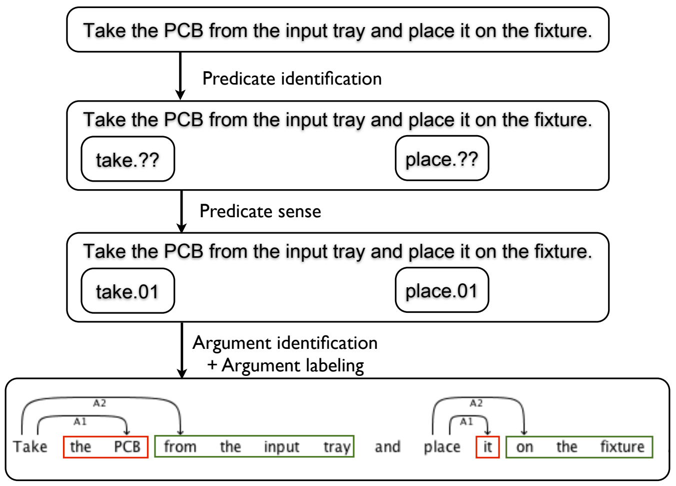

Take the PCB from the input tray and place it on the fixture.
Predicate identification
Take the PCB from the input tray and place it on the fixture.
take.??
place.??
Predicate sense
Take the PCB from the input tray and place it on the fixture.
take.01
place.01
Argument identification
+ Argument labeling
A2
A1
Take the PCB from the input tray and
A2
A1
place it on the fixture

Fig. 4. The parsing pipeline.

is extracted. The dependency graph connects the words in the sentence using their grammatical functions. It is technically a tree, where the root is the *dominant* word in the sentence, most often a verb describing an action, and the arrows (see for example bottom part of Fig. 6) point from the *parent* or *head* to its *children*. Then the predicates are identified, labelled with a sense and finally the arguments are identified and labelled. *Take* in our example has sense 1. The predicate *take.01* has three arguments named *A0-A2*, the actor (*A0*), the thing being taken (*A1*) and the source (*A2*). In this case, the robot is not explicitly mentioned, hence there is no *A0*. Pronouns, such as *it* or *them* are linked to their antecedents in the sentence.

[1] The parser is available as open source software, freely accessible at `http://barbar.cs.lth.se:8081/`.

Previous work (Stenmark and Nugues, 2013) defined an algorithm describing how predicates can be mapped to robot skills, and arguments linked to specific world objects in order to create an executable sequence of the task, as displayed in Fig. 2. However, the supported programming features were limited, excluding e.g., such control structures as conditionals, temporal constraints, control parameters, parallell execution and references to program features. The contributions of this work are that predicate-argument pairs can be mapped to complex skills and the novel methods we are using to extract constraints and control structures from NL instructions.

Code Generation and Execution The executable code for primitive actions is generated in native controller language (RAPID). E.g., each gripper can have a predefined native code to open and close it. On the other hand, the sensor-controlled skills use a framework based on iTaSC (De Schutter et al., 2007), together with external force/torque sensors. These skills are specified by state machines using JGrafchart (2012) language, where states are simple motions and transition conditions are, e.g., timeouts or force and torque thresholds. A motion is specified by constraining outputs (e.g., positions or force values) from a *kinematic chain*. The kinematic chain is a specification of the relation between task variables and the robot, which are represented by a list of transformations. The state machine is generated by ES for all skills and all constrained motions (Stenmark and Stolt, 2013).

4. PATTERN-MATCHING ALGORITHM

In this section, we present our method of extracting motion constraints and control structures from unstructured English in more detail. At the moment, the system supports cardinality, parallel execution, conditionals and program references. The algorithm that runs on KIF server is presented in Algorithm 1. It matches the output from the semantic parser to program statements, using the semantic labels, the part of speech (POS) tags and dependency relations between the words. The following examples illustrate how the matching of the different statements is carried out.

Cardinality refers to the number of elements. In the sentence *Take all needles and put them in the pallet*, the cardinality of the needles is *all*. *Take three of the needles ...* has cardinality three. The cardinality is easily extracted from the arguments. In these examples, the arguments A1 to *take.01* are *all needles*, and *three of the needles*, respectively. In the first case, the verb is labelled as plural (NNS) and the determiner *all* is used. In the second case, where there is an explicit numbering (CD) in the argument, it is used as cardinality. Personal pronouns, such as *them* or *it*, are assumed to refer to all the objects in the previous argument (this is done in the ES). There is a subtle difference between *Take the needle* and *Take a needle*, which is expressed in the use of determiner. In the first case, a specific needle is referenced, while in the second, it is only the object type that is mentioned and any needle can be chosen. When linking entities to specific objects in the world, the system will look for a specific

Algorithm 1: Pattern-matching algorithm. Non-trivial functions are described separately.

```
Data: Input text $text$, set of predicates that have an
      action-mapping, $understoodPredicates$
Result: list of program statements, list of unknown statements
Let $sentences$ be a list of sentences in $text$ split by ".", "!" and "?"
Let $actions$ be an empty list
Let $unknownStatements$ be an empty list
$sentenceNbr \leftarrow 0$
foreach sentence s in sentences do
    Increase $sentenceNbr$
    $semOutput \leftarrow$ semParse($s$)
    $q \leftarrow$ sortPredicates($semOutput$)
    while q is not empty do
        $p \leftarrow$ poll first element in $q$
        if not(p is negated or an auxiliary verb) then
            if understoodPredicates does not contain p then
                $stm \leftarrow$ createArgs($p$,$q$)
                Add $stm$ to $unknownStatements$
                $wildcard \leftarrow$ getWildcard($p$)
                if wildcard found then
                    Add $wildcard$ to $unknownStatements$
                end
            end
            else
                $stm \leftarrow$ createArgs($p$, $p$)
                Add $stm$ to $actions$ with $sentenceNbr$
                $wildcard \leftarrow$ getWildcard($p$)
                if wildcard found then
                    Add $wildcard$ to $actions$ with $sentenceNbr$
                end
            end
            Remove nested predicates in $stm$ from $q$
        end
    end
end
return actions and unknownStatements
```

Function sortPredicates(*semOutput*)

```
if semOutput has a root element then
    Let $q$ be an empty queue
    $root \leftarrow$ get root predicate from semOutput
    if root is a predicate then
        Add $root$ to $q$
        Parse the tree breath first adding all predicates to $q$
    end
end
else
    $predicates \leftarrow$ all predicates from $semOutput$ in input order
    Add all $predicates$ to $q$
end
return $q$
```

object where the name matches the argument value in the first case, but in the second case, the argument value is an object type and the system will return objects of the given type instead. When the cardinality of an argument is larger than one, the resulting program structure is a loop, the sentence number is used to determine its scope, where actions in the same sentence are in the same loop. *"Take all needles. Put them in the pallet."* will thus be two loops, and in a single robot system the planning service will complain about such instructions.

Until is a keyword for extracting the exit condition. *Until* is used to express guarded motions such as *Search*

Function createArgs(p)

```
args $\leftarrow$ findArgs($p$)
$stm \leftarrow (p, args)$
if hasIfCondition(p) then
    $word \leftarrow$ the child of $p$ of form "if" or "when"
    $condition \leftarrow$ recursiveSearch($word$)
    $stm \leftarrow$ if-statement with $condtion$ and $stm$
end
if hasBreakCondition(p) then
    $word \leftarrow$ the child of $p$ of form "until"
    $condition \leftarrow$ recursiveSearch($word$)
    $stm \leftarrow$ break-statement with $condtion$ and $stm$
end
if hasParallellActivity(p) then
    $word \leftarrow$ the child of $p$ of form "while"
    $condition \leftarrow$ recursiveSearch($word$)
    $stm \leftarrow$ while-statement with $a$ and $stm$
end
return $stm$
```

Function findArgs(p)

```
$a_1 \leftarrow$ argument "A1" of $p$
if $a_1$ does not exist then
    $a_1 \leftarrow$ search for an argument labelled "TMP", "IN",
    "AM-LOC"
    if $a_1$ is not found then
        $a_1 \leftarrow$ search among children to $p$ labelled "LOC"
    end
end
$a_2 \leftarrow$ argument "A2" in $p$
if $a_2$ is not found then
    $a_2 \leftarrow$ search for an argument labelled "TMP", "IN",
    "AM-LOC"
end
if $a_1$ is not found and $a_2$ is found then
    $a_1 \leftarrow a_2$
    $a_2 \leftarrow$ void
end
return ($a_1$, $a_2$)
```

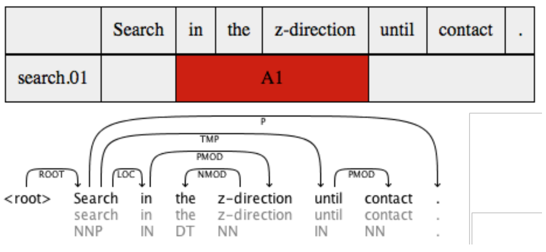
Function recursiveSearch(w)

```
foreach child c of word do
    if c is predicate then
        $cond \leftarrow$ createArgs($c$)
        if any child cc to c has POS-tag "CC" then
            $nestedStm \leftarrow$ recursiveSearch($cc$) ($cc$ is "and" or "or")
            Add $nestedStm$ to $cond$
        end
    end
end
return $cond$
```

Function getWildcard(p)

```
$manner \leftarrow$ get argument from $p$ with tag "AM-MNR"
if manner found then
    $word \leftarrow$ recursively search all descendants of $manner$ for a
    word labelled "NN", "NNS" or "NNP"
    if word found then
        $stm \leftarrow$ new statement("use", $word$)
        return $stm$
    end
end
return empty statement
```

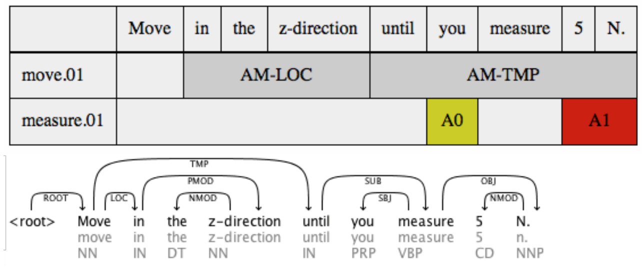
in the z-direction until contact. The conditions can be nested PA structures as well, for example: *Move in the z-direction until you measure 5 N.* The results from the parsing of the two example sentences are displayed in Figs 5 and 6. In order to extract the program statements, the analysis starts with the root in case the root is a predicate. If the predicate belongs to the set of *understood predicates*, it is added as a program statement, together with its arguments. In the first example, the direction was identified as argument *A1* to *search.01*, however, in the second sentence, the direction is considered a location argument to *move.01*. In the case of missing object arguments, the location arguments are used instead, since these are valid parameters to motions. The default frame of the direction is the tool frame.



	Search	in	the	z-direction	until	contact	.
search.01		A1					

P
TMP
PMOD
ROOT
LOC
NMOD
PMOD
<root> Search in the z-direction until contact .
search in the z-direction until contact .
NNP IN DT NN IN NN .

Fig. 5. The parse result of *"Search in the z-direction until contact"*, together with the dependency graph.



	Move	in	the	z-direction	until	you	measure	5	N.
move.01		AM-LOC			AM-TMP				
measure.01						A0			A1

TMP
PMOD
ROOT
LOC
NMOD
SUB
SBJ
OBJ
NMOD
<root> Move in the z-direction until you measure 5 N.
move in the z-direction until you measure 5 n.
NN IN DT NN IN PRP VBP CD NNP

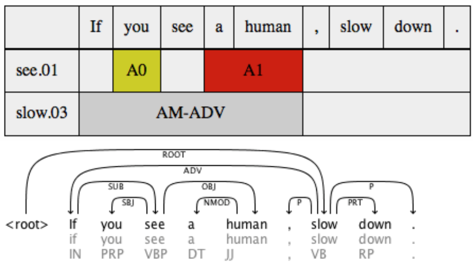
Fig. 6. The parse result of *"Move in the z-direction until you measure 5 N"*.

If the predicate has any temporal constraints, expressed by for example *until* and *while*, these are labelled *TMP* in the dependency graphs. The temporal constraints can be either a noun describing an event, or nested PA structures such as *measure* (pred) *5 N* (*A1*). The temporal constraints are added as a condition to the main program statement (*Move - z-direction*) and will later be used to create transition conditions and thresholds for the guarded motion. Conditions will be discussed in more detail later.

While. In most programming languages *while* statements are equivalent to *until*, however, in natural language they also express parallelism. For example *"While holding 5 N in z-direction, search in x-direction until contact"* is a guarded motion along one axis, while adding a constraint in another direction. The result is translated into program statements similarly to *until*-statements. This sentence results in a *while*-statement describing the parallell actions of searching and holding, while the search is a nested until-statement with the transition condition.

Conditions. Conditions can be events or PA structures. In our system, the events that can be used are *contact*, *collision* and *timeout*. The predicates that are allowed are limited to *measure*, *reach*, *sense*, thus limiting the expressions to sensor values. The system also supports nested conditions using AND and OR, such as *contact or timeout*, because *and* and *or* are tagged as coordination conjunctions (*CC*) by the dependency parser.

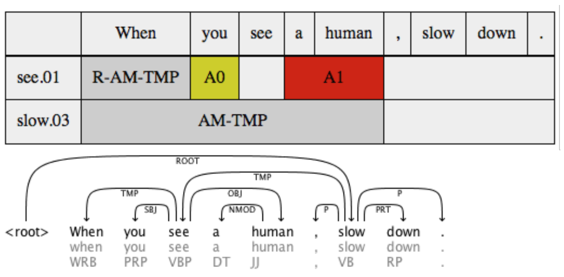
If and **when.** In our system, these are considered equivalent, however, in the if-sentence the condition is considered an adverbial while in the when-sentence it is a temporal, see Fig. 7 and Fig. 8. This difference is ignored and the PA structure is used as a condition in both cases.



	If	you	see	a	human	,	slow	down	.
see.01		A0		A1					
slow.03	AM-ADV								

ROOT
ADV
SUB
OBJ
SBJ
NMOD
P
P
PRT
<root> If you see a human , slow down .
if you see a human , slow down .
IN PRP VBP DT JJ , VB RP .

Fig. 7. Result for an if-sentence.

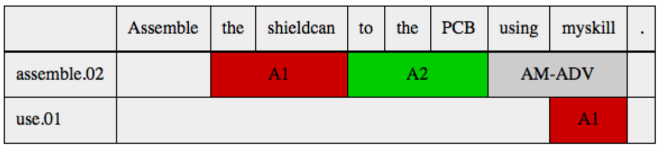


	When	you	see	a	human	,	slow	down	.
see.01	R-AM-TMP	A0		A1					
slow.03	AM-TMP								

ROOT
TMP
TMP
OBJ
SBJ
NMOD
P
P
PRT
<root> When you see a human , slow down .
when you see a human , slow down .
WRB PRP VBP DT JJ , VB RP .

Fig. 8. Result for a when-sentence.

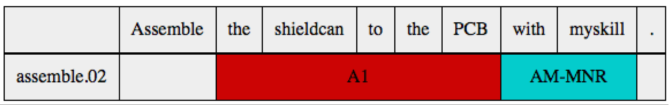
Keywords. All robot skills are not suited to be mapped to predicates, e.g., in a *Snapfit* skill two plastic pieces are snapped into position. Hence, the predicate *use* is dedicated as a keyword, where the argument is either another program or a device that is not part of the assembled parts, such as sensors or tools. That allows sentences such as *"Assemble the shieldcan and the PCB using myskill"*, see Fig. 9. Here *myskill* can be *snapfit* or *peg-in-hole*, or be replaced with tool such as *gripper2*. When a *use*-predicate is evaluated by the system, it first searches among the sensors and tools for devices that the skill can use, and then online for a skill which can be used to replace the generic *assemble* action.



	Assemble	the	shieldcan	to	the	PCB	using	myskill	.
assemble.02		A1		A2			AM-ADV		
use.01								A1	

Fig. 9. Example of the usage of the wildcard word *use*.

Another way to express similar commands is by using the word *with*. This will naturally not be parsed into a predicate, but rather be an argument to *assemble.2* called *manner* which is labelled *AM-MNR* in the result shown in Fig. 10. Adverbs typically describe the manner of a predicate, such as *Carefully assemble...*. In case the



	Assemble	the	shieldcan	to	the	PCB	with	myskill	.
assemble.02		A1					AM-MNR		

Fig. 10. Example of the usage of the word *with.*

manner contains *with* and a noun it is simply interpreted as a *use* with the noun as its argument.

Program references. A small set of predicates and PA structures are used to describe the program itself. For example *Repeat the task.* The predicates are *pause*, *stop*, *start*, *repeat*, and *restart*, while the arguments can be skills or general references such as *the task* and *the program.*

Negation. Predicates with negation are ignored. Although it is possible to imagine commands such as *Don't go close to the human*, we have chosen to require usage of an active command such as *Avoid the human.* For a negation to be meaningful, both an action and its negation have to be mapped to different skills, since the complement of an action is not a well defined concept.

When the program statements have been extracted from English sentences, the predicates are mapped into programs and functions, and the arguments are linked to objects in the world or to skills that are downloaded to the station. Thresholds for sensor values and parallell constraints are added to the guarded motions. Executable robot code for the task is generated from the guarded motions and skills. The resulting code has been verified by virtual robot execution in the Engineering System.

5. DISCUSSION

Using the standard predicate argument-structures together with the dependency graphs, it is possible to extract the semantic meaning of complicated assembly task descriptions from unstructured English. The bottleneck is rather the availability of robotic skills and functionalities in the system, not the natural language understanding by itself.

In a virtual world, control parameters and sensor thresholds can be set to default values. In order to carry out robust task execution on a physical platform though, the damping and stiffness factors of the impedance controller and force signatures should be learnt for the task. The parameters to the impedance control can be learnt by experimentation, as shown by Stolt et al. (2012).

The approach and algorithms presented in this paper are not limited to just assembly tasks, or just to industrial robot task descriptions. After having completed experiments involving skill parameter learning, we plan to extend this approach to other manufacturing domains.

REFERENCES

ABB Robotics (2013). Robot studio 5.15 overview. http://www.abb.com/product/seitp327/30450ba8a4430bcfc125727d004987be.aspx.

Billard, A., Calinon, S., Dillmann, R., and Schaal, S. (2008). *Springer Handbook of Robotics*, chapter Robot Programming by Demonstration, 1371–1394. Springer.

Bischoff, R., Kazi, A., and Seyfarth, M. (2002). The MORPHA style guide for icon-based programming. In *Proc. of the IEEE Int. Workshop on Robot and Human Interactive Communication.*

Björkelund, A., Bohnet, B., Hafdell, L., and Nugues, P. (2010). A high-performance syntactic and semantic dependency parser. In *Proc. COLING 2010*, 33–36.

Björkelund, A., Bohnet, B., and Nugues, P. (2009). Multilingual semantic role labeling. In *Proc. of CoNLL-2009.*

De Schutter, J., De Laet, T., Rutgeerts, J., Decré, W., Smits, R., Aertbeliën, E., Claes, K., and Bruyninckx, H. (2007). Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. *Int. J. Rob. Res.*, 26(5), 433–455.

Hägele, M., Nilsson, K., and Pires, J.N. (2008). *Springer Handbook of Robotics*, chapter Industrial Robotics, 963–986. Springer Verlag.

IEC (2003). IEC 61131-3: Programmable controllers – part 3: Programming languages. Technical report, International Electrotechnical Commission.

JGrafchart (2012). http://www.control.lth.se/Research/tools/grafchart.html.

MacMahon, M., Stankiewicz, B., and Kuipers, B. (2006). Walk the talk: Connecting language, knowledge, action in route instructions. In *Proceedings of AAAI 2006.*

Matuszek, C., Cabral, J., Witbrock, M., and DeOliveira, J. (2006). An introduction to the syntax and content of Cyc. In *Proc. AAAI Spring Symp. on Formalizing and Compiling Background Knowledge*, 44–49.

Palmer, M., Gildea, D., and Kingsbury, P. (2005). The proposition bank: an annotated corpus of semantic roles. *Computational Linguistics*, 31(1), 76–105.

Ruppenhofer, J., Ellsworth, M., Petruck, M.R.L., Johnson, R., and Scheffczyk, J. (2010). FrameNet II: Extended theory and practice. Tech. report, UCB.

Stenmark, M. and Malec, J. (2013). Knowledge-based industrial robotics. In *Proc. 12th Scandinavian Conference on Artificial Intelligence*, 265–274. IOS Press.

Stenmark, M. and Nugues, P. (2013). Natural language programming of industrial robots. In *Proc. International Symposium of Robotics.* Seoul, Korea.

Stenmark, M. and Stolt, A. (2013). A system for high-level task specification using complex sensor-based skills. In *RSS 2013 Workshop, Programming with constraints: Combining high-level action specification and low-level motion execution.* Berlin, Germany.

Stolt, A., Linderoth, M., Robertsson, A., and Johansson, R. (2012). Adaptation of force control parameters in robotic assembly. In *10th International IFAC Symposium on Robot Control.* Dubrovnik, Croatia.

Tellex, S., Kollar, T., Dickerson, S., Walter, M.R., Banerjee, A.G., Teller, S., and Roy, N. (2011). Understanding natural language commands for robotics navigation and mobile manipulation. In *Proceedings of AAAI 2011.*

Tenorth, M., Nyga, D., and Beetz, M. (2010). Understanding and executing instructions for everyday manipulation tasks from the WWW. In *Proc. IEEE ICRA.*

Thomas, B.J. and Jenkins, O.C. (2012). RoboFrameNet: Verb-centric semantics for actions in robot middleware. In *Proc. IEEE ICRA.*

Winograd, T. (1971). Procedures as a representation for data in a computer program for understanding natural language. Technical report, MIT.

WordNet (2010). Princeton University "About WordNet.". `http://wordnet.princeton.edu/`.