

# Instructing Industrial Robots Using High-Level Task Descriptions

**Maj Stenmark**



Licentiate Thesis, 2015

Department of Computer Science  
Lund University

# Instructing Industrial Robots Using High-Level Task Descriptions

Maj Stenmark

Department of Computer Science  
Lund University



**LUNDS UNIVERSITET**  
Lunds Tekniska Högskola

ISSN 1652-4691  
Licentiate Thesis 1, 2015

Department of Computer Science  
Lund University  
Box 118  
SE-221 00 Lund  
Sweden

Email: [maj.stenmark@cs.lth.se](mailto:maj.stenmark@cs.lth.se)  
WWW: [http://cs.lth.se/maj\\_stenmark](http://cs.lth.se/maj_stenmark)

Typeset using L<sup>A</sup>T<sub>E</sub>X  
Printed in Sweden by Tryckeriet i E-huset, Lund, 2015  
©2015 Maj Stenmark

# Abstract

With more advanced manufacturing technologies, small and medium sized enterprises can compete with low-wage labor by providing customized and high quality products. For small production series, robotic systems can provide a cost-effective solution. However, for robots to be able to perform on par with human workers in manufacturing industries, they have to become flexible and autonomous in their task execution and swift and easy to instruct. This will enable small businesses with short production series or highly customized products to use robot coworkers without consulting expert robot programmers. The objective of this thesis is to explore programming solutions that can reduce the programming effort of sensor-controlled robot tasks. The robot motions are expressed using constraints, and a number of simple constrained motions can be combined into a robot *skill*. The skill can be stored in a database together with a semantic description, which enables reuse and reasoning. The main contributions of the thesis are 1) development of ontologies for robot devices and skills, 2) a user interface that provides programming support for task descriptions in unstructured natural language and 3) an implementation where low-level code is generated from the high-level descriptions. The resulting system greatly reduces the number of parameters exposed to the user. These parameters are described on a semantic level, which means that the same skill can be used on different robot platforms. The research is presented in four peer-reviewed papers. The first covers knowledge-based instruction and the system architecture. The two following papers describe the natural language programming feature of the system as well as a description of the user interface. The fourth and last paper describes the code generation step, thus connecting the high-level language instructions to real-time executable code.



# Acknowledgements

I owe my deepest gratitude to my supervisor Prof. Jacek Malec for his continuous support during my PhD study, for his patience, encouragement and knowledge. I am also indebted to my co-supervisor Klas Nilsson, whose energy and many ideas is a great inspiration. This Licentiate thesis would not have been possible without the effort and contributions from my colleagues in the PRACE and ROSETTA projects, it has been an honor for me to work with you. I also want to thank my coworkers at the RobotLab and at the Computer Science department for their invaluable help, feedback and many fascinating discussions. Not to forget their ability to keep a positive attitude during late night demo preparations.

I also want to thank my family and friends for their emotional support and for proofreading my papers, especially Jonas Linder for his  $\text{\TeX}$ -support. Finally, I want to thank Daniel Tegnered, for an inspiring collaboration in an ongoing longitudinal psychological study and for providing material for this thesis.

*Maj Stenmark  
Lund, February, 2015*



# List of Publications

## List of Included Publications

The thesis is based on the following publications:

- Paper I** Maj Stenmark and Jacek Malec. **Knowledge-Based Instruction of Manipulation Tasks for Industrial Robotics.** *Robotics and Computer-Integrated Manufacturing*, vol. 33, pages 56–67, 2015. DOI: 10.1016/j.rcim.2014.07.004.
- Paper II** Maj Stenmark and Pierre Nugues. **Natural Language Programming of Industrial Robots.** *In Proc. of The 44th International Symposium on Robotics*, Seoul, South Korea, 2013. DOI: 10.1109/ISR.2013.6695630
- Paper III** Maj Stenmark and Jacek Malec. **Describing constraint-based assembly tasks in unstructured natural language.** *In Proc. of The 19th IFAC World Congress*, pages 3056–3061, Cape Town, South Africa, 2014. DOI: 10.3182/20140824-6-ZA-1003.02062.
- Paper IV** Maj Stenmark, Jacek Malec and Andreas Stolt. **From High-Level Task Descriptions to Executable Robot Code.** *IEEE Intelligent Systems' 2014, Series Advances in Intelligent Systems and Computing*, vol. 323, pages 189–202, Springer, 2015. DOI: 10.1007/978-3-319-11310-4\_17.

## Other Scientific Contributions

Maj Stenmark, Jacek Malec, Klas Nilsson, Anders Robertsson. **On Distributed Knowledge Bases for Small-Batch Assembly.** *IROS 2013 Workshop on Cloud Robotics*, Tokyo, Japan, 2013. <http://roboearth.org/wp-content/uploads/2013/03/final-13.pdf>.

Maj Stenmark and Jacek Malec. **A Helping Hand: Industrial Robotics, Knowledge and User-Oriented Services.** *IROS 2013 Workshop on AI Robotics*, Tokyo, Japan, 2013. [http://robohow.eu/\\_media/workshops/ai-based-robotics-iros-2013/paper07-final.pdf](http://robohow.eu/_media/workshops/ai-based-robotics-iros-2013/paper07-final.pdf).



Maj Stenmark and Jacek Malec. **Knowledge-Based Industrial Robotics**. *Frontiers in Artificial Intelligence and Applications*, vol. 257: *Twelfth Scandinavian Conference on Artificial Intelligence*, pages 265–274, IOS Press, 2013. DOI: 10.3233/978-1-61499-330-8-265.

Maj Stenmark. **Industrial Robot Skills**. *Frontiers in Artificial Intelligence and Applications*, vol. 257: *Twelfth Scandinavian Conference on Artificial Intelligence*, pages 295–298, IOS Press, 2013. DOI: 10.3233/978-1-61499-330-8-295.

Maj Stenmark and Andreas Stolt. **A System for High-Level Task Specification Using Complex Sensor-based Skills**. *RSS workshop on Programming with constraints: Combining high-level action specification and low-level motion execution*, Berlin, Germany, 2013. [http://robohow.eu/\\_media/meetings/4-stenmark.pdf](http://robohow.eu/_media/meetings/4-stenmark.pdf).

# Contents

<b>I</b>	<b>Background</b>	<b>1</b>
<b>1</b>	<b>Introduction</b>	<b>3</b>
1.1	Objectives . . . . .	4
1.2	Research Projects . . . . .	4
1.3	Thesis Contributions . . . . .	4
1.4	Thesis Outline . . . . .	6
<b>2</b>	<b>Introduction to Robot Software and Systems</b>	<b>7</b>
2.1	Automation and Robot Programming Languages . . . . .	7
2.2	Sensing and Acting in the Real World . . . . .	9
2.3	Graphics and Simulation . . . . .	10
2.4	Natural Language . . . . .	12
2.5	Databases and Ontologies . . . . .	15
2.6	Skills and Knowledge Representation . . . . .	18
2.7	Learning from Demonstration . . . . .	20
2.8	Robotic Middleware . . . . .	22
2.9	iTasC . . . . .	24
2.10	Code Generation . . . . .	25
<b>3</b>	<b>Conclusions</b>	<b>29</b>
<b>II</b>	<b>Papers</b>	<b>33</b>
	<b>Paper I: Knowledge-Based Instruction of Manipulation Tasks for Industrial Robotics</b>	<b>35</b>
1	Introduction . . . . .	37
2	Robot Skills . . . . .	37
3	Architecture . . . . .	41
4	Knowledge Integration Framework . . . . .	44
5	Knowledge-Based Services . . . . .	46
6	Engineering System . . . . .	50
7	Execution . . . . .	51
8	Related Work . . . . .	51

---

9	Conclusions . . . . .	54
<b>Paper II: Natural Language Programming of Industrial Robots</b>		<b>57</b>
1	Introduction . . . . .	59
2	Related Work . . . . .	59
3	System Overview . . . . .	60
4	High-level Programming Prototype . . . . .	62
5	Conclusions . . . . .	63
6	Future Work . . . . .	65
7	Acknowledgments . . . . .	65
<b>Paper III: Describing Constraint-Based Assembly Tasks in Unstructured Natural Language</b>		<b>67</b>
1	Introduction . . . . .	69
2	Related Work . . . . .	69
3	Background . . . . .	70
4	Pattern-Matching Algorithm . . . . .	73
5	Discussion . . . . .	79
<b>Paper IV: From High-Level Task Descriptions to Executable Robot Code</b>		<b>81</b>
1	Introduction . . . . .	83
2	System Overview . . . . .	83
3	Code Generation . . . . .	85
4	Experiments . . . . .	91
5	Related Work . . . . .	92
6	Conclusions and Future Work . . . . .	93
7	Acknowledgments . . . . .	93

**Part I**

**Background**



# Chapter 1

## Introduction

In 2014, the European Commission launched Horizon 2020, a research and innovation program. The goal is to bring new technologies to the market, to increase the industrial competitiveness of the European Union and battle societal challenges. The program lists Advanced Manufacturing as one key enabling technology. With more advanced manufacturing technologies, small and medium sized enterprises can compete with low-wage countries by providing customized and high quality products. Traditional industrial robot systems are not suitable for flexible production in close cooperation with human workers. Furthermore, conventional robot programming is time consuming and non-trivial, especially for users working in small businesses who are unfamiliar with robots. For customized products in small series, it is paramount that the user interaction and programming is swift and painless. The challenge is to develop robot systems with streamlined robot programming and robust and autonomous task execution. Hence, we see the birth of a new generation of intelligent industrial robots. These robots have 1) better communication capabilities, for example understanding human language, 2) better situation awareness with more advanced sensor integration such as object recognition, 3) better reasoning capabilities so that they can adapt their task during execution and 4) the ability to learn task parameters so that they can optimize the execution.

The work included in this thesis only addresses a few of the many challenges in intelligent robotics, namely, simplifying and automating programming of assembly tasks. The domain is small parts assembly, such as consumer electronics. The basic idea is that the human coworker should be able to reuse robot programs that are created by more advanced users and that the system should help the user to set up the task. The programming should resemble inter-human communication, that is, be goal-oriented and use actions and objects in high-level descriptions.

We have developed a framework to describe robot programs, so called skills, that can be stored in an online database, downloaded and reused using a graphical user interface. The tasks can be described using English sentences and objects in a CAD model of the world. Thus far we have limited our approach to industrial robotics and manufacturing. Other areas such as service or health care robotics face similar issues when it comes to interaction and sharing knowledge and skills between robots.

---

## 1.1 Objectives

The challenge is to build a robotic system, where the effort to program complex assembly tasks is lowered in order to make the system usable for non-expert users. Simultaneously, the system should lower the effort for more advanced users such as system integrators. This involves the creation of knowledge bases where robot skills and tasks are stored for reuse as well as services that help the user to schedule the task and generate code for sensor-based motions. The objective is to develop a complete toolchain, including user interfaces for programming and services that aid the user to refine the task for deployment.

## 1.2 Research Projects

The research work was funded by several European and Swedish research projects, and each paper includes a more detailed acknowledgement section on the matter. The most substantial are the grants from the European Union seventh framework program (FP7/2007-2013), grant agreements No. 230902 (project ROSETTA), No. 285380 (project PRACE), as well as No. 287787 (project SMErobotics).

## 1.3 Thesis Contributions

The three main contributions of the included papers are the following:

1. **Ontologies for describing robot devices and skills.** The first paper describes the ontologies and provides an overview of the system architecture. The ontologies are modular and separated into smaller sub-ontologies that are loaded into a core ontology. The core ontology includes robot devices, such as robots, sensors, tool changers and fixtures. Tasks and skills are represented by graphs, hence we developed a graph ontology which includes different state machine descriptions. Skills can be represented by a small sub-ontology that describes the device requirements and pre-and post-conditions.
2. **A service for natural language programming of robots.** The system includes distributed reasoning services that were developed in parallel with the knowledge representation framework. The services can be accessed from a high-level programming interface. Paper II describes the initial approach that uses a general purpose natural language tool to extract the semantic meaning from sentences written in unstructured English text. The extracted semantic structures are then mapped to actions in the skill database and objects in the robot workspace. The natural language programming interface was further developed in Paper III to be able to generate repeated actions (loops) and constraints for sensor-controlled motions.

- 
3. **Generation of executable code for sensor-controlled skills.** Being able to express a task using high-level semantic descriptions is not enough, it is necessary to be able to synthesize executable code as well. The initial implementation of the code generation service is presented in Paper IV.

To provide an overview, the abstracts of the papers are included below.

## **Paper I – Knowledge-Based Instruction of Manipulation Tasks for Industrial Robotics**

**Abstract.** When robots are working in dynamic environments, close to humans lacking extensive knowledge of robotics, there is a strong need to simplify the user interaction and make the system execute as autonomously as possible, as long as it is feasible. For industrial robots working side-by-side with humans in manufacturing industry, AI systems are necessary to lower the demand on programming time and system integration expertise. Only by building a system with appropriate knowledge and reasoning services can one simplify the robot programming sufficiently to meet those demands while still getting a robust and efficient task execution.

In this paper, we present a system we have realized that aims at fulfilling the above demands. The paper focuses on the knowledge put into ontologies created for robotic devices and manufacturing tasks, and presents examples of AI-related services that use the semantic descriptions of skills to help users instruct the robot adequately.

## **Paper II – Natural Language Programming of Industrial Robots**

**Abstract.** In this paper, we introduce a method to use written natural language instructions to program assembly tasks for industrial robots. In our application, we used a state-of-the-art semantic and syntactic parser together with semantically rich world and skill descriptions to create high-level symbolic task sequences. From these sequences, we generated executable code for both virtual and physical robot systems. Our focus lies on the applicability of these methods in an industrial setting with real-time constraints.

## **Paper III – Describing constraint-based assembly tasks in unstructured natural language**

**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 [83] 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.

## **Paper IV – From High-Level Task Descriptions to Executable Robot Code.**

**Abstract.** For robots to be productive co-workers in the manufacturing industry, it is necessary that their human colleagues can interact with them and instruct them in a simple manner. The goal of our research is to lower the threshold for humans to instruct manipulation tasks, especially sensor-controlled assembly. In our previous work we have presented tools for high-level task instruction, while in this paper we present how these symbolic descriptions of object manipulation are translated into executable code for our hybrid industrial robot controllers.

### **1.4 Thesis Outline**

The thesis is organized as follows: the first part is an introduction to robot programming languages and tools, first focusing on high-level programming environments and instruction methods, then providing an overview of low-level solutions for robot control, architecture and code generation. This part is concluded with a discussion, where reflections on the research work and future challenges are presented. The second part consists of four peer-reviewed publications. The first paper is intended to give an overview of the system and to shed light on the knowledge engineering aspects. The second and third papers describe the natural language programming interface, which was improved over the years. The last paper covers the code generation service.

## Chapter 2

# Introduction to Robot Software and Systems

Robot systems consist of a heterogenous mix of hardware and software components. The hardware can consist of robot manipulators, sensors, fixtures, grippers, tools and tool changers, and input devices such as teach pendants or tablets. The software can for example be image processing algorithms, control software, databases, and graphical programming user interfaces. The architecture is distributed and setting up the communication and data management of the full system while ensuring real-time requirements can be quite a challenge. In order to abstract away the details of the low-level system components, and make the software development easier, the communication and data management can be handled by middleware. The middleware can function as a virtual machine or operating system, so that the user can write robotics applications without specifying every detail of the communication between the subsystems, see Section 2.8 for more details on software architectures. In this background section, robot programming languages and tools will be introduced, followed by control software implementations.

### 2.1 Automation and Robot Programming Languages

Many industrial robot vendors have created their own programming language. For example, the market leader in industrial robotics, Motoman from Yaskawa Electric uses INFORM [100], KUKA uses KRL [45] and ABB developed RAPID [27]. These languages are designed with motion specification in mind, allowing debugging by stepping through instructions one by one, either forward or backwards. An example of RAPID code is displayed in Fig. 2.1. It is a small program where the robot arm moves to a target position (`target_3`), opens the gripper, moves to a pick up position (`target_5`), closes the gripper and retracts. The wrist is close to a singularity, hence a *singularity area* is turned on before and off after the movements. `MoveL` is a linear move (similarly, there is a joint move `MoveJ`) that moves the robot to predefined target positions (`target_3`, `target_5`, `target_6`) at

```

PROC main()
  SingArea\Wrist;

  MoveL target_3,v100,z50,tool0\WObj:=wobj0;
  IF DOutput(do1)=0 OR DOutput(do2)=1 THEN
    Reset do2;
    Set do1;
    WaitTime 0.5;
  ENDIF
  MoveL target_5,v100,fine,tool0\WObj:=wobj0;
  IF DOutput(do2)=0 OR DOutput(do1)=1 THEN
    Reset do1;
    Set do2;
    WaitTime 0.5;
  ENDIF
  MoveL target_6,v100,z10,tool0\WObj:=wobj0;

  SingArea\Off;
ENDPROC

```

Figure 2.1: An example of a small program written in RAPID.

velocity 100 mm/s. There are two output signals, do1 and do2 that are used to open (do1 is high) and close (do2 is high) the gripper.

These languages are widespread but non-standard. There are five standard languages for industrial automation applications, defined in the IEC-61131-3 standard [36]. Three of them are graphical: Sequential Function Charts (SFC), Ladder Diagram (LD) and Function Block Diagram (FBD) while Instruction List (IL) and Structured Text (ST) are textual. An example SFC is displayed in Fig. 2.2. It is created using a tool called JGrafchart [91], which is developed and used in Robot-Lab at Lund University. The squares are states or steps in the state machine; the initial state is marked with double borderlines. A transition is green if it is true (update) and triggers the following states. Here two parallel states are active until cond becomes true. The last state, with marked corners, is a *macrostep*, a nested state machine.

There are several graphical tools for programming of state machines, e.g., ROS, the Robot Operating System [73], has the state machine editor *Smach* [79] or behaviors in *Bride* [17], or, the representation can be textual, such as StateMachines in Orocos [70].

However, more advanced applications, such as multirobot cooperation and sensor controlled tasks, are not easily programmed in native robot code or executed locally on a robot controller. Hence, software and middleware intended to simplify integration and package distribution use standard programming languages. For example, the KUKA Sunrise control system uses Java [44], ROS uses Python and C++ and Orocos [70] uses C++.

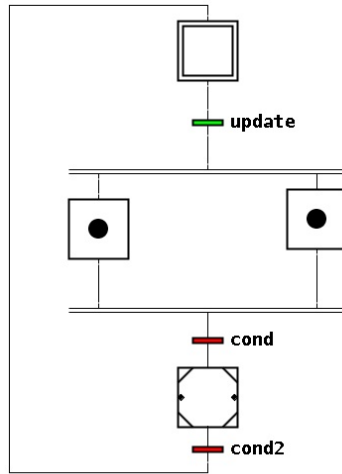


Figure 2.2: An example of an SFC.

## 2.2 Sensing and Acting in the Real World

Robots have high repeatability, which means that they can repeat the same motion very precisely multiple times. This is useful in settings where the environment also is repeatable, that is, in a robot cell where workpieces are placed in fixtures with known positions. Flexible production and a dynamic environment add another level of complexity to the robot programming. Depending on the application, some robots are equipped with cameras and detection algorithms for object localization, and may reason about their tasks during execution, for example, calculate how to grip an object in a corner or navigate around an obstacle.

In Fig. 2.3 a dual-arm mobile robot is shown. A two-armed ABB industrial robot mounted on top of a mobile platform. Down to the right, one of the yellow laser scanners may be glimpsed; scanners are used for mapping and obstacle avoidance. The mobile robot platform runs ROS, where off-the-shelf packages for mapping and navigation are available for download and use. On top of the robot a *teaching handle* is attached. The balls are tracked by a camera system and the human user can demonstrate positions and position areas for the robot; see more about learning from demonstrations in Section 2.7.

Sensor-controlled assembly tasks can be expressed using constraints on the sensor values. For example, a constraint may require that the robot should keep a constant pressure on a surface while moving over it at the same time. Tasks like these can be expressed using the iTaSC formalism, described further in 2.9.

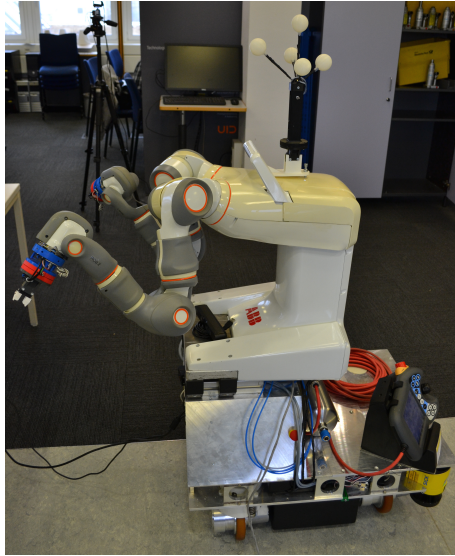


Figure 2.3: A two-armed ABB industrial robot mounted on top of a mobile platform. On top of the robot a teaching handle is attached.

## 2.3 Graphics and Simulation

A robot program can be debugged and optimized using graphical simulation environments. Typically, such simulation environments have a graphical representation of the robot and the workspace, and a virtual robot controller that can execute the instructions in a realistic manner. Some environments, such as Gazebo [64], which can be used in ROS, have physics simulations (such as light conditions or simple gravity). There are commercial environments such as 3DCreate from Visual Components [96] that are intended for factory design and simulation and support several robot brands. Usually, robot vendors provide tailored off-line programming software and virtual robot controller simulation software as well as task specific packages for their products. Examples of such vendor specific software are: MotoSim[55] that comes with painting application packages, Fanuc’s ROBOGUIDE [23] with packages for, e.g., painting, welding and palletizing, KUKA.Sim from KUKA robotics and ABB RobotStudio for simulation of ABB robots.

In this thesis, a simplified programming tool was developed as an extension to ABB RobotStudio. Sequences of robot instructions are composed into tasks, as seen in Fig 2.4. The RobotStudio extension, in the included papers called the Engineering System, provides a *natural language programming interface*, where the user can input sentences in English that are translated into a sequence of instructions (see Papers II and III). Short introductions to natural language processing are given in the papers, but a more comprehensive background is provided below.

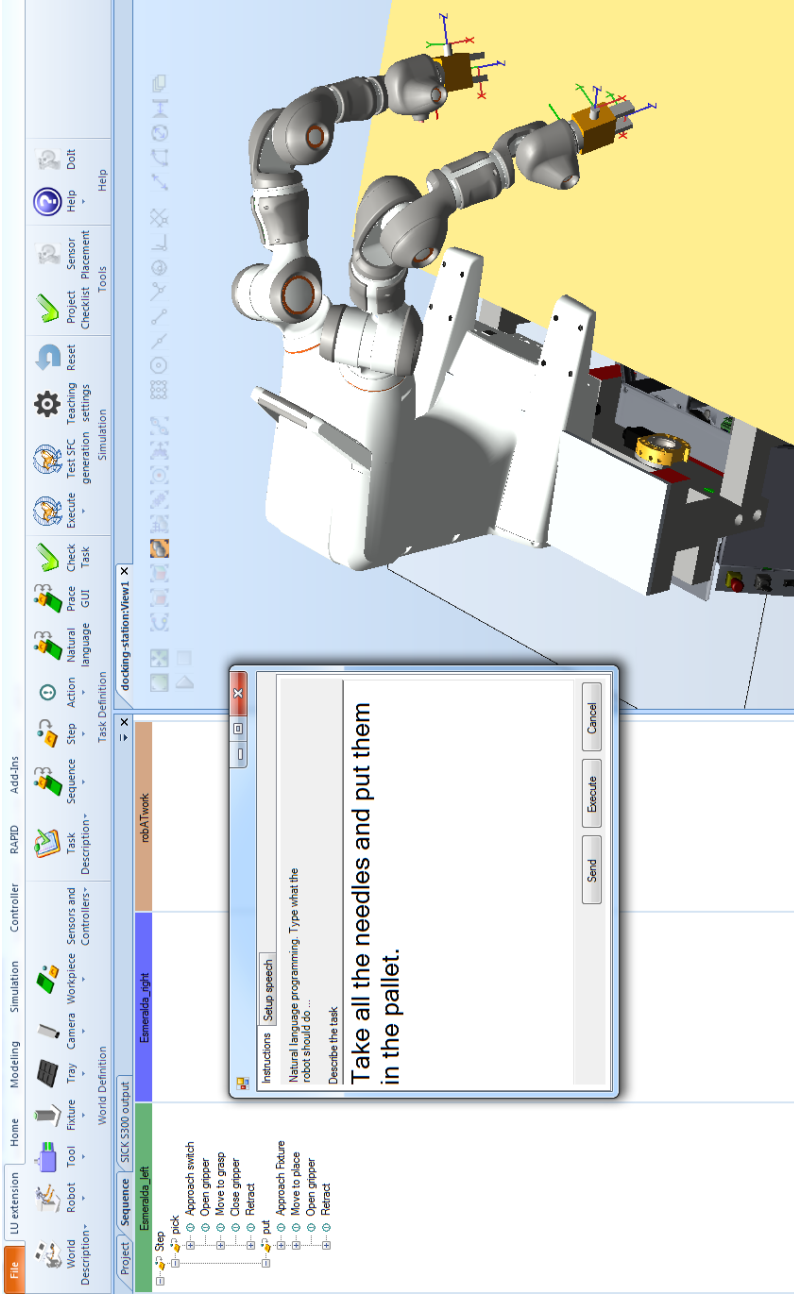


Figure 2.4: The LU Extension is a plugin to the graphical programming environment ABB RobotStudio. The extension provides natural language programming and simple sequencing of actions and skills into tasks, as seen to the left.

---

## 2.4 Natural Language

The idea of using natural (human) language to instruct a machine is older than Unix time itself. A famous early attempt is SHRDLU [99] developed by Terry Winograd. SHRDLU was a computer program that understood English and let a user describe how to move objects in a block world. This was in 1968 and the program could understand 50 words, such as "block", "cone", "blue", "place on" and tried to guess the user's intention from the word combination. An introduction to natural language processing (NLP) applications is given by [62]. The first approaches were ruled-based, where hand-made grammatical rules were used to process text [63]. However, for unrestricted natural language a large number of rules is needed, especially for handling homophones, ambiguous interpretations and spoken language. Hence, the turn of the century saw the rise of statistical NLP, where large annotated text samples were used to train statistical models. In statistical approaches, a model is learnt from a training set and then used to categorize the most likely meaning of a sentence. In the 21st century the power, performance and availability of natural language processing and speech recognition systems rapidly increased due to increased processing power, data driven approaches and Internet services. The general purpose statistical model running on a virtual machine at Lund University can process (parse) unstructured English sentences in about 20 ms and is accessed from for example mobile devices. The result from the parser is a *semantic* structure of the input sentence.

Semantics is the study of meaning. When a meaning of a sentence is extracted the following questions are at least partially answered:

- What is going on? (Predicate)
- Who/what is doing it? (Actor)
- What objects are involved? (Arguments to the predicate)
- Where is this going on? (Location argument)
- When and how long is this going on? (Temporal argument)
- How is the action carried out? (Manner)
- Who is the beneficiary of the event? (Beneficiary)

Language is ambiguous, homophones, such as *press* in *news press* and *press down* belong to different part-of-speech and have different meaning. In a sentence, the words that answer the above questions are labeled with the *semantic role* listed in parenthesis after the question. This is done using special purpose dictionaries such as WordNet [69], FrameNet [76], PropBank [66], that list predicates and roles that can belong to the predicate. For example, in the sentence *I live in a cardboard box*, the verb *live* is the predicate. There are several versions of *live* in PropBank, so called *senses* that are different flavors of the word. The most common meaning of *live*, *live.01*, is to reside or to not being dead. Another example is *live.02*, to endure. In the example sentence, the predicate has an actor entity *I* and a location argument *in a cardboard box*.

---

The goal for the Lund semantic parser and role labeler is to read the sentences and output predicate-argument structures and present them in a table as seen in Fig. 2.5. The figure shows the parsed output from the sentence *I have found the perfect Chicago-style deep dish pizza recipe*<sup>1</sup>, an unusual sentence containing a long compound noun. Two predicates are found, *find.01* and *recipe.01*, each corresponding to a line in the top table where the semantic roles belonging to the predicates are marked. The pronoun *I* is the actor *A0* and, since modifiers and determiners are included in the argument, *the perfect Chicago-style deep dish pizza recipe* is the argument *A1* to *find.01*. The semantic description of *recipe.01* contains a *theme* as *A1*-argument, in this case, there are three individually labeled *A1* arguments. The graph in the middle is the dependency graph, which in fact is a tree starting with the *root* of the sentence. The arrows are labeled with the grammatical function between the words, such as modifiers (NMOD), subject (SBJ) or object (OBJ) relations to the verb *have*, which is also the root. At the bottom of Fig. 2.5, a table displaying the words in CoNNL-2009 standard is shown.

Determining the predicate-argument structures of a sentence is a classification problem in machine learning. The model for the parser is trained using a *corpus*, which is a large text mass where the sentences are annotated with syntactic and semantic information. There are several corpora, for example the Penn Treebank [49] which is created from *Wall Street Journal* articles, and the PropBank [66] that added predicate-argument structures to the Penn TreeBank. The latter is used by the Lund parser.

The Lund parser uses three main steps to classify the sentence, further described in [10]. First the sentence is split into words (tokenized), and each word is assigned a part-of-speech tag and the canonical form (*lemmatization*). Next, it produces a *dependency graph*, which is a graph structure with the grammatical relations between the words, see Fig. 2.5 for an example. The dependency graph is finally used in a semantic role labeler to produce the predicate argument structures. The semantic role labeler uses binary or multiple logistic regression in a cascade of classifying steps.

- **Predicate Identification:** Each word is either classified as a predicate or not using binary logistic regression.
- **Predicate Disambiguation:** Determines the sense of the predicate if there are multiple senses. Lemmas that can be both verb and nouns have one classifier per part-of-speech.
- **Argument Identification:** First each word is either classified as an argument or not. Then a multi-class identifier determines the role of the word.

As long as the sentence is somewhat grammatically correct, the parser will produce a well-formed machine-readable table structure. From this table, the output can be matched to robot instructions from a *skill database* and arguments can be mapped to objects in the world, as described in Papers II and III.

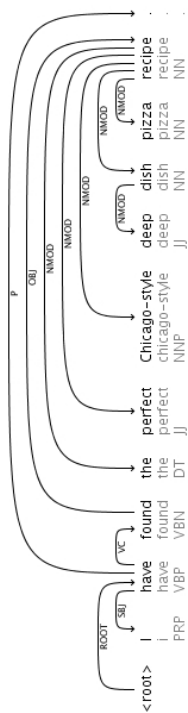
---

<sup>1</sup>It contains two pounds of mozzarella.



	I	have	found	the	perfect	Chicago-style	deep	dish	pizza	recipe	.
find.01	A0				A1						
recipe.01	A0				AM-MNR	A1	A1	A1	A1		

Parsing sentence required 34ms.



ID	Form	Lemma	PLemma	POS	PPOS	Feats	PFeats	Head	PHead	Deprel	IsPred	Pred	Args: find.01	Args: recipe.01
1	I	i		PRP					2	SBJ			A0	
2	have	have		VBP				0	ROOT					
3	found	found		VBN				2	VC	Y	Y	find.01		
4	the	the		DT				10	NMOD					
5	perfect	perfect		JJ				10	NMOD					
6	Chicago-style	chicago-style		NNP				10	NMOD					AM-MNR
7	deep	deep		JJ				8	NMOD					A1
8	dish	dish		NN				10	NMOD					A1
9	pizza	pizza		NN				10	NMOD					A1
10	recipe	recipe		NN				3	OBJ	Y	Y	recipe.01	A1	
11	.	.		.				2	P					

Figure 2.5: The parsed output from the example sentence I have found the perfect Chicago-style deep dish pizza recipe.

Subject	Predicate	Object	Context
<a href="http://thesis.se/maj">http://thesis.se/maj</a>	<a href="http://www.w3.org/2002/07/owl#individual">http://www.w3.org/2002/07/owl#individual</a>	<a href="http://thesis.se/PhDstudent">http://thesis.se/PhDstudent</a>	
<a href="http://thesis.se/maj">http://thesis.se/maj</a>	<a href="http://thesis.se/hasName">http://thesis.se/hasName</a>	"Maj"	
<a href="http://thesis.se/maj">http://thesis.se/maj</a>	<a href="http://thesis.se/studiesAt">http://thesis.se/studiesAt</a>	"LundUniversity"	

Figure 2.6: A screen capture from a triple store with three triples, one triple per row.

## 2.5 Databases and Ontologies

The knowledge of the system is stored in a database. During the project, the knowledge grew incrementally and the world was assumed to be *open*<sup>2</sup>, hence a relational database with static tables was not a suitable choice for the conceptual model. Instead the system used an *RDF triple store* and later, a graph database. RDF, short for Resource Description Framework, is a format where data is described as triples. Our implementation uses the Sesame Workbench [65]. Each triple  $\langle S, P, O \rangle$ , has a *subject node*  $S$ , a *predicate*  $P$  and an *object node*  $O$ . The object node can also be a primitive data type such as string or a number, called a *literal*. Literals can not point to any other nodes and unconnected edges are not allowed. Hence, a triple is equivalent to a directed edge in a graph and a node can have multiple outgoing and incoming edges.

Our graph database is implemented using Neo4J [56]. In the graph database the literals can be attached directly to the nodes, and are called *properties*, while an edge between nodes is called a *relationship*. A relationship can also have properties, which is not possible to express using RDF. Hence, a graph database can encode information in a more dense format than a triple store.

To concretize, we will now encode some simple data in RDF and a graph database, and the reader will become acutely aware of the need for an *ontology*, which will be introduced afterwards.

Assume that we want to create a small social network with some information about people<sup>3</sup>. One person is named "Maj", with the occupation "PhD student" at a university called "Lund University". In RDF this can be encoded as triples as following:

```
@prefix thesis: <http://thesis.se/>.
```

```
thesis:maj owl:individual thesis:PhDstudent ,
thesis:maj thesis:hasName "Maj" ,
thesis:maj thesis:studiesAt "LundUniversity" .
```

Resources and properties in RDF are identified using unique Uniform Resource Identifier (URIs). The URI can be split into a prefix and an ending, in the example above *http://thesis.se/* is a prefix named *thesis*. One object, *thesis:maj*,

<sup>2</sup>The knowledge is not complete.

<sup>3</sup>To store personal information in a structured format requires explicit consent from the individuals, according to *Personuppgiftslagen*.

is defined as a individual of a class *thesis:PhDstudent*, with properties labeled *thesis:hasName* and *thesis:studiesAt* with literal values. The listing above creates three triples as shown in Fig. 2.6. A resource, either a subject, an object or a property is identified with URIs, hence *http://thesis.se/maj* refers to the same node in each row. "Maj" and "LundUniversity" are string literals. In the first triple, the predicate refers to an external URI and a class *PhDstudent*. The property *owl:individual* that assigned a class type used the web ontology language OWL [97]. In a moment we will create a small ontology so that we can make some simple reasoning, but first we compare how the same information can be stored in a graph database.

In Neo4J the node could have literal properties attached on the node, hence the same information can be stored as one single node. The previous example is visualized in Fig. 2.7a, showing one data node of type *PhDstudent* with an arbitrary ID 13916 and the properties *name* and *studiesAt*. To make our social network a little bit more interesting we can add another node and a social relationship between the nodes. We add another *PhDstudent* called "Daniel", who studies at "Chalmers" and connect the nodes with a relationship, as seen in Fig. 2.7b. Relationships (written with capital letters) between nodes can have literal properties as well. Here the relationship *KNOWS* has two properties *since:2012* and *relationship:romantic*.

The relationships are directed, so when looking up persons in the database the query has to be expressed correctly, as seen in Fig. 2.8. The query (a) finds all nodes *n* that the node *maj* with the name property set to "Maj" points to with a *KNOWS* relationship. It will return the *Daniel* node. Query (b) matches nodes that *daniel* points to, with an empty result because the *KNOWS* relationship is not (yet) symmetric. Depending on the relationship type, other relationships can be inferred. For example, there might be an inverse property and when one is true the other can be inferred. In a social setting such pair could be *hasParent* - *hasChild* relationships. Another type of property is *transitive*, for example, a PhD student is also a *Person*, hence, if *Maj* is a *PhDstudent*, the system should infer that she also is a *Person*. It might also be desirable to limit the allowed number of relationships

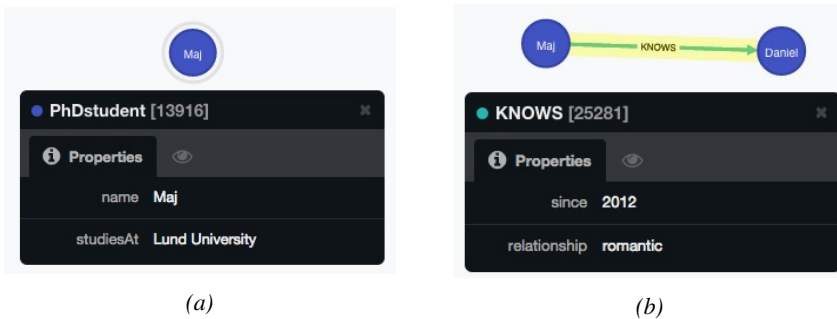


Figure 2.7: A single database node to the left and two nodes connected with a *KNOWS* relationship to the right.

```
MATCH (maj {name:"Maj"})-[:KNOWS]->(n)
RETURN n;
```

(a)

```
MATCH (daniel {name:"Daniel"})-[:KNOWS]->(n)
RETURN n;
```

(b)

Figure 2.8: The query finds all nodes  $n$  that the node "Maj" knows (a) and queries to find who "Daniel" knows (b).

of a certain type, the *cardinality* of a relationship.

Modeling this type of knowledge can be done in an *ontology*. A small example is shown in Fig. 2.9. Classes are marked with a yellow circle, individuals of a class with a purple diamond. The violet properties are *is-a* relationships<sup>4</sup>, while blue is a subclass relationship, and brown dashed lines are used for all other properties. Here there are individuals *Maj*, *Daniel*, *Lund\_University* and *Chalmers*. Each student has to have a *studiesAt* relationship to some university. After running an inference engine on the ontology in Fig. 2.9, new relationships are added as shown in Fig. 2.10. Class properties are inferred for both instances of *PhDStudent*, and since the relationship between *Maj* and *Daniel* is a symmetric property, an arc between the nodes is added.

When an ontology is created, assumptions are made about the reality, e.g., that there are only two legal parents, etc. However, in reality, there can be contradicting assertions, information sources might be dubious or faulty, properties can change over time, or our model can be too simplistic. There are different approaches to handle these inconsistencies, e.g., by adding probabilities to a statement, evaluating a statement by looking at other supporting statements, or when data is updated, by using time stamps. The approaches are very much context- and application-dependent, e.g., in a social network, the relationship status between *Maj* and *Daniel* must be verified by both parties, contracts or by legal records.

In the context of robotics, ontologies are used to represent knowledge about robots, sensors, and tasks. The working group *Ontologies for Robotics and Automation* has developed a Core Ontology for Robotics and Automation, CORA,

<sup>4</sup>In this context, the meaning of the word 'is' is nonexclusive, one individual can belong to multiple classes.

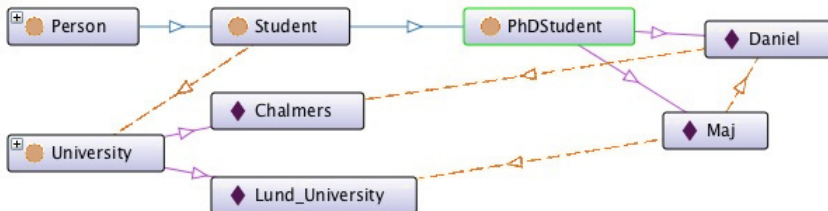


Figure 2.9: A small example ontology.

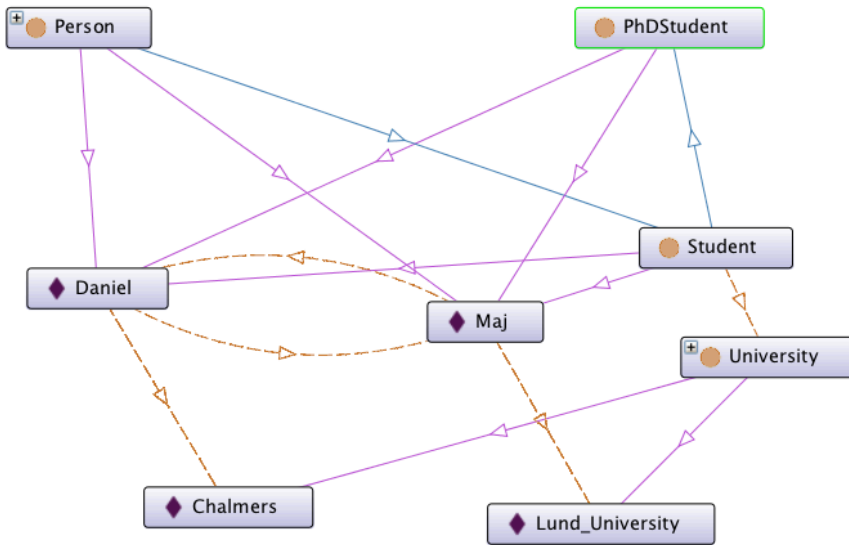


Figure 2.10: The ontology from Fig. 2.9 with inferred relationships.

defining positions [19] and later extending it with environments [26]. The group has also created an ontology for kit building [4] which has been evaluated using human and multi-robot interaction [38]. This work is complementary to the ontologies presented in Paper I that focus on tasks and devices used in industrial applications, and are used as inspiration in ongoing work. Other ontologies for common household tasks and objects were developed in the ROBOEARTH project [89]. The ROBOEARTH ontology is based on OPENCYC [46], an ambitious project to create an ontology for common sense reasoning. One challenge in robotics is how to represent task descriptions and robot programs, often called *skills*, in a sensible way.

## 2.6 Skills and Knowledge Representation

*Skills* is an overloaded term in the field of intelligent robotics. There is no universal definition of a robot skill, nor how a skill should be formally represented. The state of anarchy is so widespread, that there is no consensus on whether a skill is represented by the goals of a task<sup>5</sup> or the procedure to reach a goal.

A more established concept is *manipulation primitive*, which is the building block of skills. It is an interface between the sensor-based motion control and robot programming [43] which is based on the Task Frame Formalism [18, 21] where coordinate frames can be placed in the work cell or attached to the robot. Motions and sensor-values can then be expressed in any local coordinate frame

<sup>5</sup>Here "task" is used in its natural language sense because it, too, is a provocative term.

---

as long as it is connected to the robot in a chain of known frames. The motion primitives described in [43] and implemented by [25, 68] contain a *hybrid motion* expressed in a frame, a *tool command* which executes synchronously with the motion and a *stop command* which terminates the move. The implementation presented in Paper III is based on the same formalism and a similar three-tier architecture, however, the technical implementation details differ. The problem with the motion primitives is that the programming blocks are too low-level to add value to a system. This has resulted in the notion of skills, a combination of primitive robot capabilities that **are non-trivial and have production value**. The production value depends on how advanced the skill is and also on how *reusable* it is. Hence, there are parallel ongoing efforts to develop a viable skill concept.

The skill description closest to our work is described in [68]. Their skills consist of a sequence of motion primitives encapsulated between pre-condition, post-condition and continuous checks. The skill is initialized with a number of parameters and terminates with an evaluation of post-conditions. The emphasis on evaluation is where the skill description diverges from ours, since we do not require an evaluation step even though this can be desired. That is, in our approach, a peg-in-hole skill does not need to include, e.g., visual inspection of the inserted object. Such inspection procedure can in fact be another skill. Another difference is that their skill has to be able to estimate if the skill can be executed on the input parameters and the world state. This is not a requirement for us, where only the skill parameters have to be within allowed intervals and all pre-conditions fulfilled. It is possible to simulate the skills from the primitive sequence in our case. The more relaxed requirements stem from a pessimistic view on the simulation capabilities of the robot and its ability to evaluate the skill execution in each step. Instead, the inspection skills are added in intervals during the task. The requirements in [68] are relevant for planning and reasoning purposes, but not during the execution. The differences are academic however, since the skill designer either way decides what conditions and checks to include in the skill.

Another related approach is the Action Recipes from the ROBOEARTH [89] project. These are high-level task descriptions in an OWL-based language. The Action Recipe is closer to our task description, because it can include (partial) ordering between actions and the actions are fairly advanced (we would describe them as skills), such as handing over drinks, capabilities that need to be implemented on the robot.

In this project, a task is the overall goal of the robot. It can be a set of partially ordered subgoals. These subgoals can be represented in a graph structure, called *assembly graph*. An example of an assembly graph is shown in Fig. 2.11. It displays an assembly of an emergency stop button box. There are two partially ordered subassemblies. To the left, a red button should be pushed through a hole in a yellow box top, and then a nut should be screwed on the button from the other side of the box. To the right, another subassembly is displayed: a dark grey switch should be snapped into place on a light grey box bottom. Finally, the two subassemblies should be joined. The graph has a tree structure, where the leaves are the original workpieces while each parent node represents an assembly.

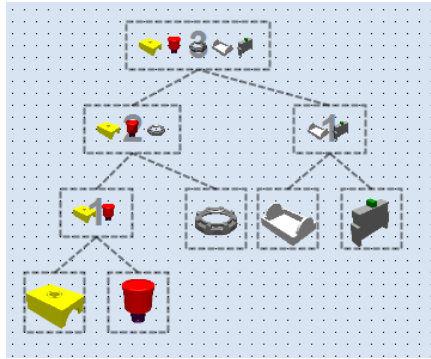


Figure 2.11: An example of an assembly graph of an emergency stop button box.

Child assemblies must be carried out before the parent assemblies, but otherwise no ordering is imposed. Each assembly node can specify which skill to use for the assembly. To execute the task, a sequencing is given to the skills by a planning or scheduling algorithm, or, if no optimization is required at that state, any valid ordering.

During this project, skills are usually implemented as sensor-based motions in form of an SFC, where actions can be generated as native robot code as described in Paper IV. The skills are stored in an RDF database together with *ontologies*, as described in Paper I. An ontology is used to structure data so that it is possible for a machine to reason about it. For example, a skill is stored online together with a set of input parameters with types, values and units. The type can be a start position with  $x$ ,  $y$  and  $z$  parameters given in mm relative to the robot world coordinate system, and rotations in Euler angles in radians or degrees. The types can be used by the graphical interface to select views and convert between units. Another example is a picking skill, where a post-condition of the skill is that an object is mounted in the gripper. This post-condition can be used by a scheduling service so that the object has to be released before another one is picked up. Also object types such as workpieces and sensors are stored in the ontology, which makes it possible to reason about the work cell.

Since the robots act in the physical world, using skills that couple perception and action, one compelling programming method is programming by demonstration. From demonstrations, parameters such as trajectory, accuracy and sensor values can be extracted from the statistical data. This is useful since parameter values that depend on the setup and can be difficult to know beforehand.

## 2.7 Learning from Demonstration

Learning from demonstration (LfD), or *imitation learning* is a method where users teach a task to a robot by physical demonstrations, without traditional programming. It is inspired by the way humans learn by imitation from young age. Until

---

recently, LfD has mostly been used for service robotics, while industrial robots used kinesthetic teach-in of positions without generalization. The development of new LfD techniques is outside of the scope of this thesis, however, it is a compelling (relatively new) robot programming paradigm that will be a large part of future work.

An introduction to the topic can be found in [2, 7] and in a more recent electronic format on Scholarpedia [8]. The benefit of LfD is that the demonstration can be carried out by lay people, and the robot can itself generalize from few demonstrations, and, if the task fails, the user can provide more demonstrations. The generalization step is crucial for LfD, thus, teaching positions and replaying the same trajectory is not LfD.

When imitating, the first thing to determine is *what* to imitate. What features are relevant in the demonstration? Is it relative or absolute positions of objects, for example? Should the robot learn how the world maps to actions or the goal of the task? Should the robot generalize to discrete symbolic actions or continuous functions? When the desired output is symbolic, such as simple actions, classification techniques such as Hidden Markov Models, Gaussian Mixture Models, Support Vector Machines, decision trees, etc., can be used. When the learned task is continuous, such as a trajectory, regression methods can be used, for example Locally Weighted Regression. In the latter case, the statistical model can use the variance of the demonstrations to determine where high precision is needed during the motion.

The second design choice is *how* to imitate. The demonstrator and the learner can differ in perceptual and physical capabilities, hence the learner will have to transform the demonstrated data to its own perception and joint space. For example, the demonstration can be carried out by directly observing the human using vision or sensor gloves. This is simple for the human but it can be difficult to find corresponding robot behavior. Another approach is kinesthetic teaching, where the human moves the manipulator into place. It makes the imitation simpler for the robot, but the demonstrator might need two arms to move the robot, hence making it nearly impossible to demonstrate two-armed robot motions. Such limitations can be addressed by teaching different parts of the task incrementally. The user can use haptic devices to teach the task remotely, however, it can be non-trivial to control a robot with multiple degrees of freedom using a remote control [8].

Demonstrations are naturally limited by the ability of the teacher and the human-robot teaching interface, hence, it is desirable that the robot improves beyond the capabilities of the teacher. LfD can be coupled with reinforcement learning techniques to improve the performance. In that case, the teacher must provide a reward function or the robot must learn the reward function itself. Creating a good reward function is non-trivial.

All the above-mentioned methods are application-dependent. One challenge is to reuse learned skills on a different robot and use adaption algorithms to optimize the task on the new platform. Another is that the teaching methods require background knowledge and engineering, such as a known reward function, which can be application-dependent and thus not easily be transferred to another task.



---

## 2.8 Robotic Middleware

Robotic systems are built by a number of distributed heterogeneous hardware and software components that have to interact during execution. In a robotic system such components can include sensors (e.g., force sensors, laser scanners and cameras), actuators and different software modules (control algorithms, motion planning, etc). In order to simplify configuration, communication and hide the complexity of the system, as well as promote portability and modularity, there are several frameworks for robotic *middleware*. Middleware is an abstraction layer between software applications and the operating system and provides an interface for execution and communication between components. Hence, middleware can decrease development time and simplify code reuse. Surveys of current robotic middleware are presented by [22, 54], a few are mentioned below.

### 2.8.1 ROS

The Robot Operating System, ROS [73], is a robotic middleware that runs on Ubuntu and it is quite popular in the research community. Code modules called *nodes* can be written in Python or C++. The nodes communicate using an asynchronous publisher-subscriber model or by calling blocking services on other nodes. In the publisher-subscriber model there are a number of *topics* that publisher nodes can write to and other nodes can subscribe to. A publisher node can be a sensor that publishes data messages, for example, a camera node with an image message. A Master node helps the nodes to set up the communication. The publisher node will advertise its topics to the Master, and a subscriber node, for example an image processing algorithm, connects with a subscribe call to the Master which sets up direct communication between the nodes. The topics are one-directional and asynchronous, but there are also synchronous *services* where one node sends a request and a response is returned.

Since ROS has gained popularity, there is a large collection of open source packages for sensors, robots, navigation with varying levels of quality. However, in the industry, the enthusiasm has been mild, since the flat architecture gives scaling issues and the system lacks real-time guarantees. One attempt to cater to the needs of the industry is *ROS industrial* [74], however the initiative is yet in its infancy.

In RobotLab in Lund, ROS is used for example on the mobile robot platform in Fig. 2.3. The ROS system communicates with the Orca system running on the robot and the task level state machine in JGrafchart using LabComm-ROS bridges, see Section 2.8.3.

### 2.8.2 OROCOS

The Open Robot Control Software, OROCOS [70], is a framework for real-time robot control, thus complementary to ROS. Similarly to ROS the software is organized in modules, here called *components*. However, the OROCOS Real-Time Toolkit is designed with hard real-time control in mind, letting the user to de-

---

termine scheduling and periodicity of components and the component designers must enforce real-time behavior. Hence, making two independently written components work together can be difficult, a hardship that ROS users can ignore. Other frameworks, such as Rock [72] build on the OROCOS toolchain provide additional features such as monitoring tools.

### 2.8.3 LabComm

Middleware can simplify integration and software reuse within a robot system, but it can make it difficult for two robot systems to communicate even if both run the same middleware. Hence, it is important to have a neutral communication protocol as well. LabComm is a communication protocol developed at Lund University. Each message is typed and the user specifies the data format in a text file, a *sample*, and from that sample a *LabComm compiler* generates encoders and decoders in different programming languages, e.g., C, RAPID, Java or C#. A LabComm sample can either be a single type such as the *status* integer below,

```
sample int status;
```

It can also be a struct, as the *test\_sample* below. Here, *list\_of\_floats* is an array of floats, the length is specified as an integer between the brackets, or with an underscore for arrays of variable length.

```
sample struct{
    int blah;
    float list_of_floats[_];
} test_sample;
```

Using LabComm, a system running Java, such as JGrafchart, can communicate for example with a system running RAPID, such as the native robot controller from ABB and an external controller as shown in Fig. 2.12. To communicate with a ROS system, the LabComm sample can be translated to a ROS message using a *LabComm-ROS bridge* that generates a small ROS node that works as message server. An overview of the software architecture used in this work is shown in Fig. 2.12. The task execution is carried out in JGrafchart, which in turn communicates with the Engineering System, ExtCtrl and the native controller using the protocol LabComm. Before and after the execution, Java processes on the Linux machine can communicate with the Knowledge Integration server and Engineering System in order to load newly generated state machines or upload log data. Before execution, the MATLAB/Simulink model that contains the control program and the sensor signals is loaded into ExtCtrl. The external controller, *ExtCtrl* communicates with JGrafchart using Orca, a layer on top of LabComm where the samples are either *input* or *output* signals, *logdata* or *parameters*. The messages from JGrafchart to ExtCtrl are the output signals, the input signals go in the other direction. Input signals are for example sensor values. Parameters and output signals are used to set up the values for the *kinematic chain*, which are described in the following Section.



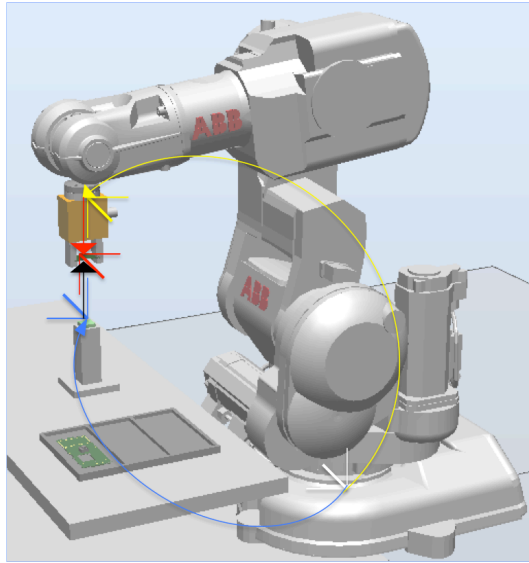


Figure 2.13: An example of a kinematic chain.

## 2.10 Code Generation

In order to go from high-level task descriptions to low-level motion control, we need to generate code for the low-level system. The first implementation is described in Paper IV.

The task is created in RobotStudio as a sequence of actions, where each action is mapped to a step in the JGrafchart state machine, as illustrated in Fig. 2.14. Simple motions are coded in RAPID, where positions are stored as RAPID robot targets and move commands are written in procedures that can be called over LabComm, e.g. from the task state machine in JGrafchart. Multiple guarded motions (with the same kinematic chain) are bundled into a nested SFC, while native code function calls are made to code loaded into the native controller. Reused skills are loaded from the knowledge-base and also put in a nested SFC. Between the generated steps the controllers are turned on and off and parameters reset.

In the task state machine, sensor-controlled motions are encoded as macro steps. The initial step in the macro step sets up the kinematic chain. The constraints for the chain are specified in RobotStudio as follows: the robot type can be ABB YuMi right or left, IRB120 or IRB140. The frames are given by six coordinate values, position values in millimeters in  $x$ ,  $y$  and  $z$  direction, and rotation in EulerYZX in radians. These values are set as parameters to ExtCtrl and used to specify the kinematic chain in the first step as shown in Fig. 2.15. The first step sets the parameters to the kinematic chain, in this case for the right arm (the first index of the `kinematicsConverged` array becomes 1). The second step is a search in  $z$ -direction. In this step, reference values for velocity and force are set along dif-

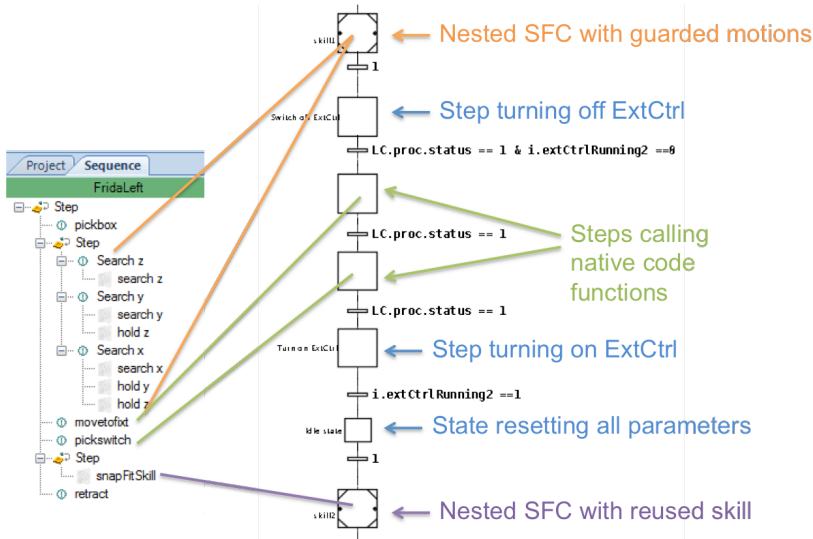


Figure 2.14: During the code generation step, a state machine is created from a RobotStudio sequence.

ferent axes and the controller parameters are set (damping and stiffness in different directions). The transition condition contains two final constraints, either the force value in  $z$ -direction is greater than 3.0 N or larger than 6.0 N in  $y$ -direction on the sensor called `y_meas_extR`.

After the kinematic chain is specified, the numerical values for the Jacobians are computed iteratively, and `kinematicsConverged` is set to true (1).

A search step is specified setting a constant velocity or force along each coordinate axis, where the values are set in a reference array to the controller. Controller parameters are set up together with the reference array, specifying the damping and stiffness values corresponding to the constant force reference value. The final constraint is set as a transition condition on the sensor values, in Fig. 2.15 the force/torque sensor on the right arm is an array named `y_meas_extR`.

Integrating a ROS system, for example, is similar to using RAPID, function calls are made using a LabComm bridge that runs in ROS. When switching between controllers, the robot has to stand still so that the controller, `ExtCtrl`, `RAPID` or otherwise, can set the reference position and velocity values to the actual values of the robot.

The implementation described in Paper IV only supports sequences and guarded motions. Ongoing work involves *loops*, *error handling* and *two armed motions*. For loops, the use case has been *palletizing*, where a pallet has a grid of positions which can be specified in several ways e.g., by two diagonal endpoints

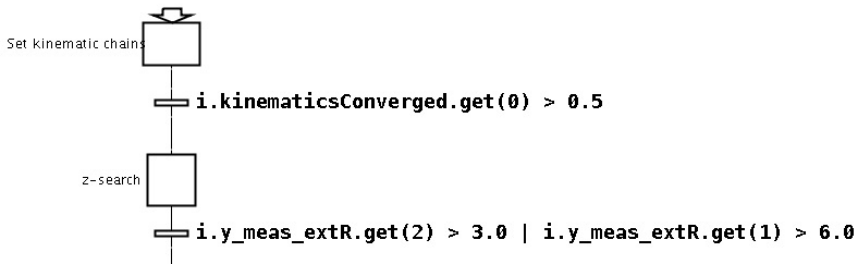


Figure 2.15: The nested SFC.

and number of rows and columns in the grid. When moving objects, in our use case needles, between two pallets, two such grids must be specified and as well as offsets between the needle and the grid point. In a more general case, where the task is to carry out multiple assemblies of objects, e.g., boxes, the grid is not specified by a pallet directly, but by the positions of the objects in pallets. On the state machine level, it is straight-forward to represent lists of positions, but the user interaction is still under development.



# Chapter 3

## Conclusions

The main contribution of this work is a system for high-level programming with code generation and natural language understanding. Using high-level programming reduces the number of manually specified parameters. Table 3.1 shows a comparison between the number of parameters that the user needs to specify in the graphical user interface and the SFC in JGrafchart, respectively. In the Engineering System, there is an initial configuration of the workspace when the coordinate frames attached to sensors, tools and objects are set up. This is done graphically, hence all six coordinate values are set at once, or each value can be specified manually. Using the high-level programming interface, each search can be set up using 4 values and one terminating condition. In the state machine each search or maintenance constraint (constraining another axis than the motion direction) can be set up by setting 6 reference values and 42 control parameters for an arm with 7 joints. A move can be specified in the graphical user interface using 4–7 parameters chosen from drop-down boxes, while this would need 21 parameters in RAPID together with a call from the state machine and controller switches. As an example, using the high-level interface to program the sequence in Fig. 2.14 (ignoring the reused skill in the end) can be done with 67 parameter when all optional parameters are set. This corresponds to 393 parameters in JGrafchart, as well as additional RAPID modules and additional steps for switching controllers. Hence, we have reduced the parameters that the user has to specify to 17 % of the original number. Using the natural language programming interface and the default parameters will reduce the effort further. The three guarded motions and six constraints (that is 190 parameters) from the sequence in Fig. 2.14 are specified with approximately 30 words.

Additionally, the task specification abstracts away from robot type and sensor name, making it reusable on different robot platforms and more easily understood by a human. The modular architecture also makes it easier to change systems, for instance, to generate code for another state machine execution system than JGrafchart.

The semantic skill descriptions in KIF makes it possible to automatically check,



---

plan, and schedule tasks. A feature that is not included in the current test implementation is automatic generation of plan domain definition language (PDDL) files. Such files can be used by off-the-shelf planners as demonstrated by [3, 41].

Only a reduced part of the system has been tested in a factory-like setting. The complete system is experimental and in order to test usability, a near product-like stability should be achieved. It is possible and desirable to test isolated parts of the system with naïve users, however, this is something we have not tried yet in a methodic manner.

## Future Work

Each paper lists more detailed future work, however, there are some general comments. User studies should be carried out in order to test usability. This would require that a subset of the system functionality is isolated and a product-like stability is provided.

The software modules that have been developed should be packaged and distributed to the community, some as open source and some as commercial products. A distribution system for robotics applications is under development at Lund University.

As the system is modular, the future work includes testing how well the system can be extended with new functionality, for example, code generation modules for other low-level languages.

One of the problems with our approach is skill acquisition. To populate the database with advanced skills, more advanced users need to create and parametrize them. To avoid the possible bottleneck this can cause, one possibility is to use learning from demonstration to automatically segment and parametrize skills. This is planned for a future project<sup>1</sup>, where the system will be extended to support learning from demonstration, segmentation and parameter learning.

---

<sup>1</sup>EU Horizon 2020 SARAFun.

Feature	Number of parameters/GUI	Number of parameters/JGrafchart	Comment
Initial setup per frame	One click or 6 manual parameters	N/A	Each frame needs to be specified either graphically or manually.
Initial setup per impedance controller	2 (stiffness and damping)	N/A	The damping and stiffness factor needs to be specified. These can be different when moving in different directions.
Initial setup of communication	Automatic	Creation of LabComm object and setting up connection port and IP, and reusing 500 lines of RAPID code.	
Setting and resetting of kinematic chain	Automatic	12 parameters per frame (typically 3-4 frames), control and mass matrices (121 values per arm).	
Search motion	4 (object, frame, axis, velocity)	6 reference values and $6 \times 7$ (or 6) control parameters.	There are 6 control parameters per joint, e.g., sampling period, stiffness and damping .
Error/Final constraint	2 (value and type or axis)	1 condition	The condition will terminate the action it belongs to.
Maintenance constraint	5 (object, frame, axis, value, controller)	6 reference values and $6 \times 7$ (or 6) control parameters.	A search constraint with a maintenance constraint will only set the parameters once, maintenance constraints without a search is purely sensor controlled in that direction.
Move (primitive actions)	4 + 3 optional parameters	Would be specified in RAPID with 21 parameters and called from JGrafchart using 3 lines of code for setting parameters, sending and waiting for acknowledgement, as well as steps for switching controllers..	
Reuse of skills	Optionally 10-20 parameters, all initialized with default values	Approximately linear to the number of steps.	It is possible to reuse the state machine directly by copying it into a macro step and manually adjust values, sensor names and solve compilation errors.

Table 3.1: The number of parameters that the user will need to specify when setting up a task in the GUI or directly in a JGrafchart state machine.



**Part II**

**Papers**



# Paper I

---

## Knowledge-Based Instruction of Manipulation Tasks for Industrial Robotics

Maj Stenmark

Jacek Malec

Department of Computer Science  
Lund University  
maj.stenmark@cs.lth.se  
jacek.malec@cs.lth.se

### ABSTRACT

When robots are working in dynamic environments, close to humans lacking extensive knowledge of robotics, there is a strong need to simplify the user interaction and make the system execute as autonomously as possible, as long as it is feasible. For industrial robots working side-by-side with humans in manufacturing industry, AI systems are necessary to lower the demand on programming time and system integration expertise. Only by building a system with appropriate knowledge and reasoning services can one simplify the robot programming sufficiently to meet those demands while still getting a robust and efficient task execution.

In this paper, we present a system we have realized that aims at fulfilling the above demands. The paper focuses on the knowledge put into ontologies created for robotic devices and manufacturing tasks, and presents examples of AI-related services that use the semantic descriptions of skills to help users instruct the robot adequately.



# 1 Introduction

The availability of efficient and cheap computing and storage hardware, together with intensive research on big data and appropriate processing algorithms on one hand, and on semantic web and reasoning algorithms on the other hand, makes the existing results of artificial intelligence studies attractive in many application areas.

The pace of adoption of the knowledge-based paradigm depends not only on the complexity of the domain, but also on the economic models used and the perspective taken by the leading actors. It may be quite well illustrated by opposing the service robotics area (mostly research-oriented, mostly publicly funded, using open source solutions, acting in non-standardized and not-yet-legally codified domain) with industrial robotics (application-oriented, privately funded, using normally closed software, enforcing repeatability and reliability of the solutions in legally hard-controlled setting).

When robots are working in dynamic environments, close to humans lacking extensive knowledge of robot programming, there is a strong need to simplify the user interaction and make the system execute as autonomously as possible (but only as long as it is reasonable). This also motivates the integration of AI techniques into robotics systems. For industrial robots working side-by-side with humans in manufacturing industry, AI-based systems are necessary to lower the programming cost with respect to the required time and expertise. We believe that only by building a system with appropriate knowledge and reasoning services, we can simplify the robot programming sufficiently to meet those demands and still get a robust and efficient task execution.

In this paper, we present a knowledge-based system aimed at fulfilling the above demands. The paper is focusing on the knowledge and ontologies we have created for the robotized manufacturing domain and is presenting examples of AI-related services that are using the semantic descriptions of skills to help the user instruct the robot adequately. In particular, the adopted semantic approach allows us to treat skills as compositional pieces of declarative, portable and directly applicable knowledge on robotized manufacturing.

The paper is organized as follows: first we introduce the robot skill, then we describe the system architecture. Next section introduces our robot skill ontology and other relevant ontologies available in the knowledge base, as well as some services provided by the system. Next we introduce the interface towards the user, i.e. the Engineering System, and briefly describe the program execution environment exploiting the knowledge in a non-trivial way, then we describe the related research. We conclude by suggesting future work.

## 2 Robot Skills

Our approach is anchored on the concept of a *robot skill*. As it may be understood in many different ways, both by humans and machines, it needs to be properly defined and made usable in the context of our domain of applications. The presentation in this section adopts a historical perspective, showing how our understanding of skills pushed forward the capacities of systems we have created.

Our earliest deployed system has been developed in the context of the EU project SIARAS: *Skill-Based Inspection and Assembly for Reconfigurable Automation Systems*. Its main goal was to build fundamentals of an intelligent system, named *the skill server*, capable of supporting automatic and semi-automatic reconfiguration of the existing manufacturing processes. Even though the concept of skill was central, we have assumed *devices* as the origin of our ontology. Our idea then has been that skills are just capabilities of de-



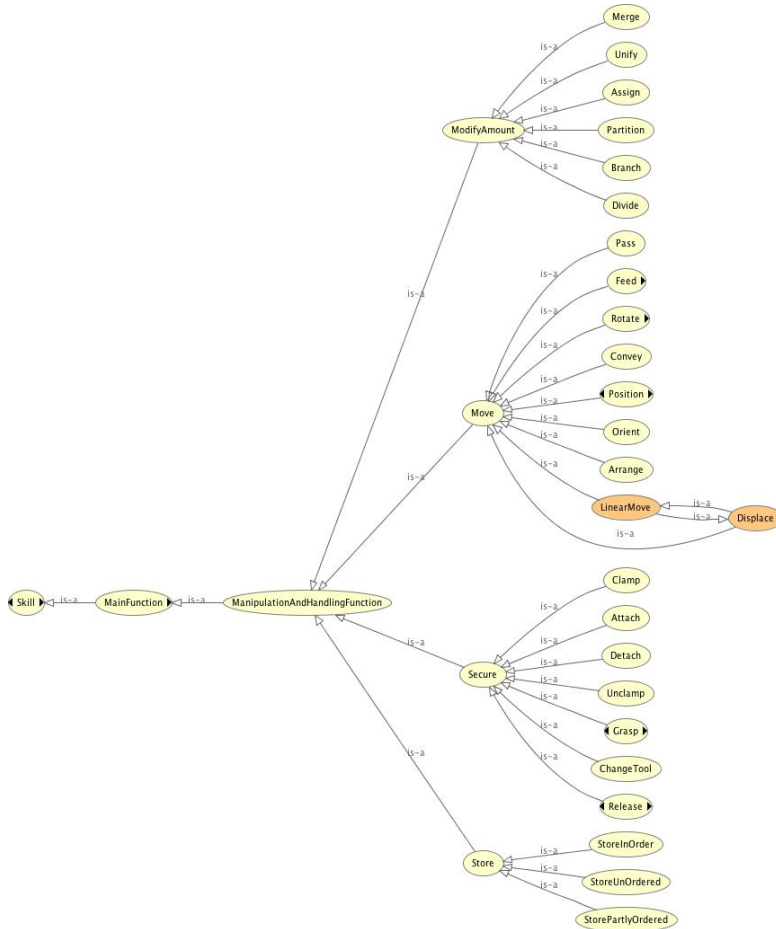


Figure 1: Manipulation and handling skills, as defined by SIARAS ontology.

vices: without them no (manufacturing) skill can exist. A device can offer one or more skills and a skill may be offered by one or more devices. We have not introduced any granularity of such distinction; all the skills were, in a sense, primitive, and corresponded to operators as understood by AI planning systems (models of operations on the world, described using preconditions, postconditions, sometimes together with maintenance conditions). This understanding laid ground to the development of a robotic skill ontology, `siaras.owl`, that has been used to verify the configurability of particular tasks given current robotic cell program expressed as a (linear) sequential function chart (SFC). This approach has been proven to be valid, but the ontology grew quite fast and became problematic to maintain, given dozens of robots with a number of variants each, thus multiplying the number of devices. The details of SIARAS approach have been described in [32]. Fig. 1 and Fig. 2 illustrate the basic hierarchy of skills available in the `siaras.owl` ontology.

The dual hierarchy, that of devices, has been illustrated in Fig. 3 and Fig. 4, while



Figure 2: Top skill classification, as defined by SIARAS ontology.



Figure 3: Manipulation and handling devices, as defined by SIARAS ontology.

Fig. 13 shows some of the properties that can be attributed to devices.

The deficiencies of the SIARAS ontology, that is, atomicity of skills and devices, fixed parameterizations and scalability issues, have led us to reconsider the idea. These time devices did not play a central role any longer, but rather skills have been put in the center. In the ROSETTA project<sup>2</sup> the definition of skills has been based on the so-called production (PPR) triangle: *product, process, resources* [20] (see Fig. 6). The *workpieces* being manufactured are maintained in the product-centered view. The manufacturing itself (i.e., the process) is described using concepts corresponding to different levels of abstraction, namely *tasks, steps, and actions*. Finally, the resources are materialized in *devices* (capable of sensing or manufacturing). The central notion of *skill* links all three views and is one of

<sup>2</sup>Robot control for Skilled ExecuTion of Tasks in natural interaction with humans; based on Autonomy, cumulative knowledge and learning, EU FP7 project No. 230902, <http://www.fp7rosetta.org/>.

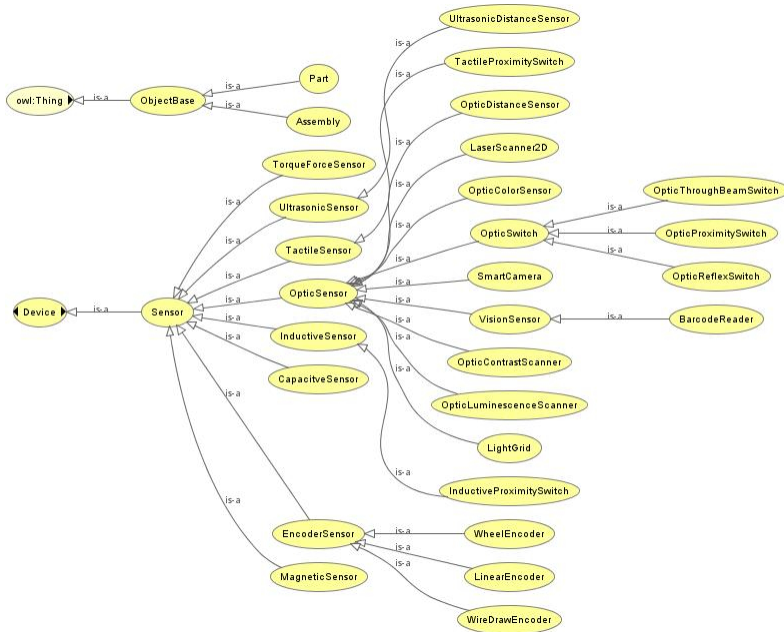


Figure 4: Sensor devices, as defined by SIARAS ontology.

the founding elements of the representation.

In case of a robot-based production system, skills may be defined as *coordination of parameterized motions*. This coordination may happen on several levels, both sequencing (expressed, e.g., via a finite state machine or a similar formalism), configuring (via appropriate parameterization of motion) and adapting (by sensor estimation). On top of this approach, based in our case on *feature frame* concept [21], we have built a set of reasoning methods related to task-level description, like, e.g., task planning. The details are presented in the following sections

### 3 Architecture

The generic setup describing the intended usage of our approach is illustrated in Fig. 7. The system architecture is very roughly depicted in Fig. 8. The Knowledge Integration Framework (KIF) is a server that contains data repositories and ontologies. It provides computing and reasoning services. There are two main types of clients of the KIF server, the Engineering System, which is a robot programming environment, and the robot task execution system.

The task execution system is a layer built on top of the native robot controller. Given the task, the execution system generates the run-time code files utilizing online code generation (see Section 5), then compiles and executes the code.

The Engineering System uses the ontologies provided by KIF to model the workspace objects and downloads skills and tasks from the skill libraries. Similarly, new objects and skills can be added to the knowledge base by the Engineering System. Skills that are cre-

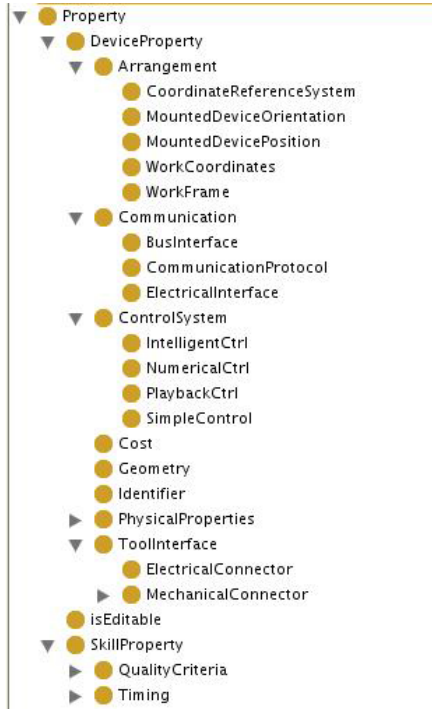


Figure 5: Device properties, as defined by SIARAS ontology.

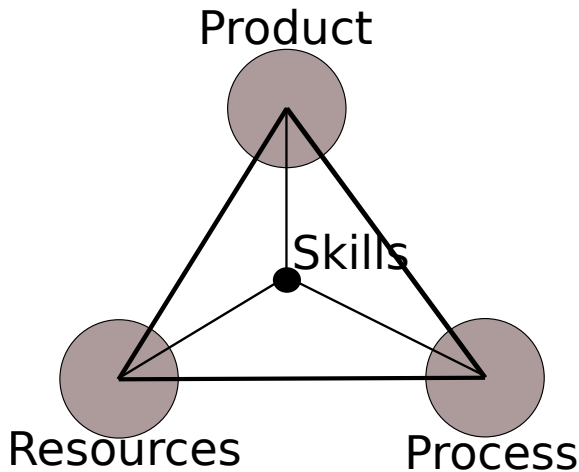


Figure 6: The PPR model, with skills as common coordinating points for the three views.

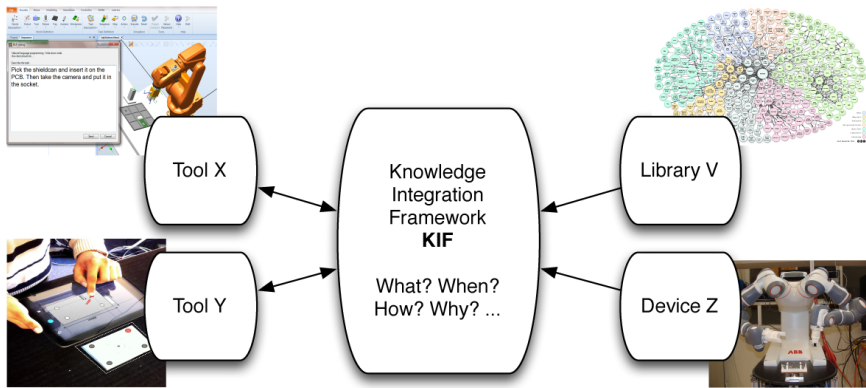


Figure 7: The Knowledge Integration Framework provides services to the Engineering System and the Task Execution. The latter two communicate during deployment and execution of tasks. The Task Execution uses sensor input to control the robot and tools.

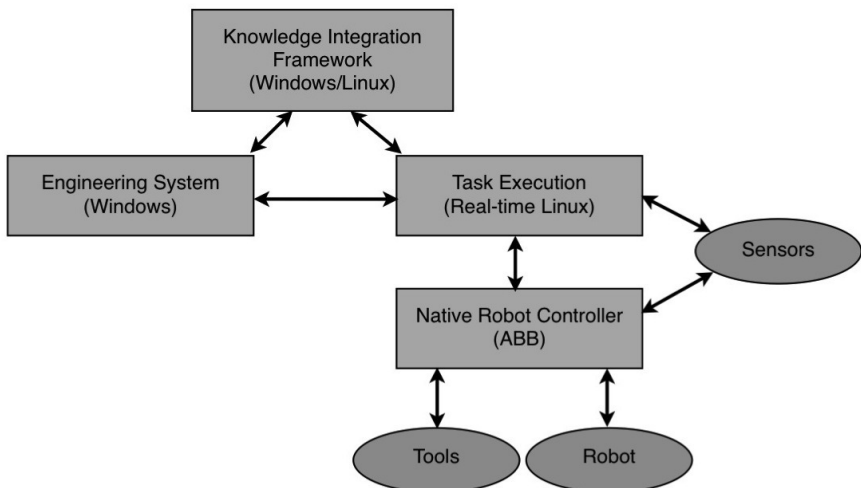


Figure 8: The Knowledge Integration Framework provides services to the Engineering System and the Task Execution. The latter two communicate during deployment and execution of tasks. The Task Execution uses sensor input to control the robot and tools.

ated using classical programming tools such as various state machine editors (like, e.g., JGrafchart<sup>3</sup> [94]), can be parsed, automatically annotated with semantic data and stored in the skill libraries.

The services, described later in the paper, are mainly used by the Engineering System to program, plan and schedule the tasks.

## 4 Knowledge Integration Framework

The Knowledge Integration Framework, KIF<sup>4</sup> is a module containing a set of robotics ontologies, a set of dynamic data repositories and hosting a number of services provided for the stored knowledge and data. Its main storage structure is a Sesame<sup>5</sup> triple store and a set of services stored in Apache Tomcat<sup>6</sup> servlet container.<sup>7</sup>

The ontologies we use in our system come from several sources and are used for different purposes. The main, core ontology, `rosetta.owl`, is a continuous development aimed at creating a generic ontology for industrial robotics. Its origins is the FP6 EU project SIARAS described earlier in Section 2. It has been further modified within the FP6 EU project RoSta (Robot Standards and reference architectures, <http://www.robot-standards.eu/>, [58]). Within the FP7 EU Rosetta project this ontology has been extended, refactored and made available online on the public KIF ontology server <http://kif.cs.lth.se/ontologies/rosetta.owl>. However, this is just the first of a set of ontologies available on KIF and useful for reasoning about robotic tasks.

The ontology hierarchy is depicted in Fig. 9, where arrows denote the ontology import operations. We used extensively the QUDT ontologies and vocabularies (Quantities, Units, Dimensions and Types, initiated by NASA and available at <http://www.qudt.org>) in order to express physical units and dimensions. This ontology has been slightly modified to suit the needs of our reasoner. However, as QUDT ontologies led to inconsistencies, we have introduced the possibility to base the quantities, units and dimensions on the alternative OM ontology<sup>8</sup> [71].

The core Rosetta ontology (as its predecessors) is focusing mostly on robotic devices and skills. According to it, every device can offer one or more skills, and every skill is offered by one or more devices. Production processes are divided into tasks (which may be considered specifications), each realized by some skill (implementation). Skills are compositional items: there are primitive skills (non-divisible) and compound ones. Skills may be executed in parallel, if the hardware resources and constraints allow it.

On top of the core ontology we have created a number of "pluggable" ontologies, serving several purposes:

**Frames** The `frames.owl` ontology deals with feature frames and object frames of physical objects, normally workpieces involved in a task. In particular, the feature frames are related to geometrical locations and therefore the representation of location is of major importance here. The constraints among feature frames are expressed using *kinematic chains* [21],

---

<sup>3</sup><http://www.control.lth.se/grafchart/>

<sup>4</sup>We realize the name coincidence with Knowledge Interchange Format [29], but as this name has been used for more than six years by now, we have decided to keep it.

<sup>5</sup><http://www.openrdf.org>

<sup>6</sup><http://tomcat.apache.org>

<sup>7</sup>Technically speaking, the triple store is also a servlet.

<sup>8</sup><http://www.wurvoc.org/vocabularies/om-1.6/>

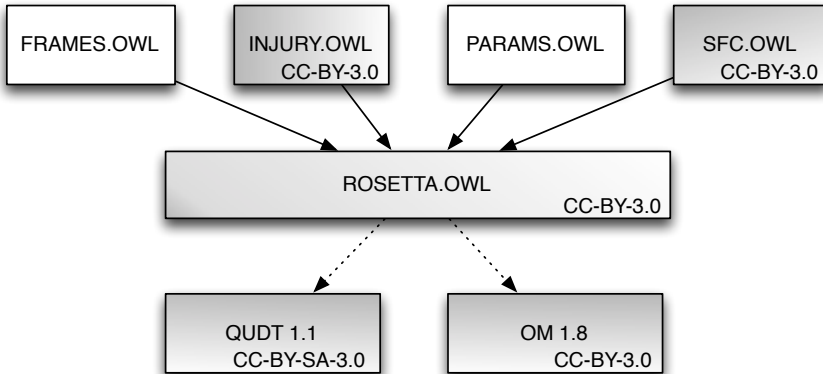


Figure 9: The KIF ontologies used by the Rosetta project. In case an ontology is openly available, the type of license is quoted.

also introduced by this ontology.

**Injury** The `injury.owl` ontology deals with the levels of injury risks when humans and robots cooperate, or at least share common space. The ontology specifies the possible injury kinds, while the associated data, either extracted from earlier work [93], or gathered during the Rosetta project [50], are provided as the upper limit values that may be used in computations of injury risks or of evasive trajectories for a robot.

**Params** Each skill may be parameterized in a number of ways, depending on the granularity level of control, available information or the demands posed on the skill. In order to provide knowledge about skill parameterization for knowledge services (like, e.g., task consistency checking), the `params.owl` ontology describes skills and their mandatory and optional parameters, their units and constraints.

**SFC** The `sfc.owl` ontology characterizes various behavior representations using variants of executable state machines (Sequential Function Charts are one of them; the others included are OpenPLC, Statecharts, rFSMs and IML). It also contains the semantic description of several graph-based representations of assembly, like assembly graphs, constraint graphs or task graphs [48], that may also be considered to be behavior specification, although at a rather high level of abstraction.

This solution illustrates two important principles of *compositionality* and *incrementality*: every non-trivial knowledge base needs to be composable out of simpler elements, possible to be created by a single designer or team without the need to align it with all the other elements. The alignment, or conflict resolution (e.g., inconsistency), should be performed (semi-)automatically, after plugging the element into the system. So, every "top" ontology should only be forced to adhere to QUDT (or OM) and ROSETTA ontologies, possibly neglecting other elements existing in parallel.



The incrementality principle ensures that every "top" ontology should be amenable to incremental change without the risk of breaking the whole system. Thus, changes to, e.g., Params ontology should not affect the consistency and utility of, e.g., SFC ontology. On the other hand, one can imagine situations where changes in one module (e.g., introduction of a new constraint type between feature frames, described in `frames.owl`) might facilitate improvements in another (e.g., easier specification of parameters for a given skill, described in `params.owl`).

Besides storing the ontologies, the triple store of KIF provides also a dynamic semantic storage used by Engineering System to update, modify and reload scene graphs and task definitions. Depending on the kind of repository used, some reasoning support may be provided for the storage functionality. More advanced reasoning, and a generic storage of arbitrary kind of data, is provided by KIF services, described below.

## 5 Knowledge-Based Services

The knowledge base provides storage and reasoning services to its clients. The most basic service it offers is access to libraries with objects and skills, where the user can upload and download object descriptions and task specifications. Some of them are stored with semantic annotations, as triples, e.g., workpieces, scene graphs or skill definitions. Others are stored as uniform chunks of data without semantically visible structure (e.g., RAPID programs or COLLADA files), although other tools may access and meaningfully manipulate them for various purposes.

The services are mostly user-oriented, providing programming aid, and can be used step-by-step to create a workspace and then to refine a task sequence from a high-level specification to low level code. The workspace is created by adding a robot, tools, sensors and workpieces to the scene, giving the object properties relevant values and defining relations between objects (see Section 6).

The user specifies a task using the workpieces and their relations. On the highest level, the task is represented by an assembly graph [48]. An example assembly graph of a cell phone is shown in Fig. 10. The assembly graph is normally a tree (not necessarily binary). The leaves are the original workpieces which are joined into subassemblies represented by parent nodes and the full assembly is represented by the root. Each subassembly can be annotated by more information, such as geometrical relations between the objects, or what type of joining mechanism to use (e.g., glueing, snapping, screwing). The tree imposes a partial order on the operations, where child assemblies have to be carried out first. When going from the task specification given by the assembly graph to an executable program, the task has to be sequentialized. Depending on the robot, or on the number of collaborating robots, the sequence can be realized in several ways, hence, an assembly graph specification can be shared by several robot systems, even though the sequences realizing it will differ.

KIF provides a planning service that transforms an assembly graph to a sequence of operations using preconditions and postconditions of the skills. Initially the service verifies the device requirements of a skill. Fig. 11 displays the device requirements of an implementation of the skill that inserts a shieldcan onto a printed circuit board (PCB). This skill has only three device requirements: a mounted tool (which is a manipulation requirement), a fixture and a force sensor, which (though it is not displayed in the figure) must be aligned vertically. When planning the sequence, the planner adds actions that fulfill the preconditions (see the example in Fig. 12), such as moving objects into place.

However, the sequence can also be created directly by the user, either manually or by

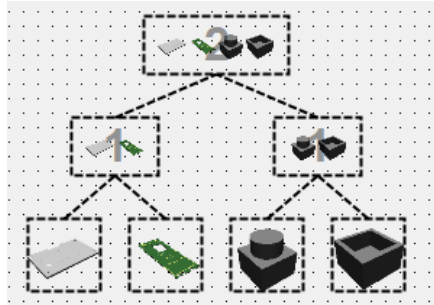


Figure 10: An assembly graph for a partial assembly of a cell phone. A metal plate, a shield can, is pressed onto a printed circuit board (PCB) and a cell phone camera is inserted into a socket. The camera socket is then fastened on the PCB.



Figure 11: The device requirements of the skill *ShieldCanInsertion* are modelled in an ontology. The **ManipulationRequirement** which several skills inherit from, is that a gripper has to be mounted on the robot. The **ShieldcanFixtureRequirement** and the **ShieldcanForceSensorRequirement** list that there must exist a fixture and a force sensor that have to be vertically aligned (not shown in the picture).



Figure 12: There are three preconditions to the *ShieldCanInsertion* skill. The skill has two feature frames (relative coordinate frames) as input parameters, where one is a reference object frame and the other is attached to the object in the gripper, i.e., an actuating frame. The first precondition is that the object with the reference frame has to be on the fixture. Secondly, the object with the actuating frame should be attached to the gripper, see Fig. 13, and finally, the position of the actuating object should be above the fixture. Imprecise geometrical relations such as “Above” are given concrete values by the Engineering System.

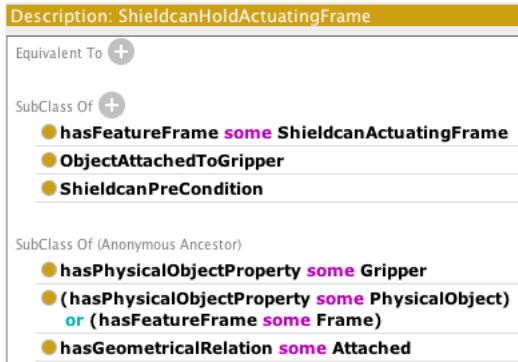


Figure 13: The ontology description of the precondition **ShieldcanHoldActuatingFrame** which is a subclass of **ObjectAttachedToGripper**.

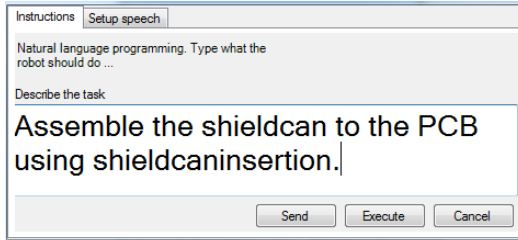


Figure 14: The user can describe the task using English sentences.

using a natural language instruction interface. Later, the same planner can be used to verify that the sequence fulfills the preconditions of each action (Fig. 13).

The natural language interface is described in more detail elsewhere [83] and [81]. The user either dictates or types English instructions in a text field (see an example in Fig. 14). The input text is sent to a natural language service on KIF where the sentences are parsed into predicates (verbs) and their corresponding arguments. Each verb has several different *senses* depending on the context and meaning, e.g., the predicate *take* in *take off the shoes* has sense *take.01*, but in the sentence *Take on the competition* it has sense *take.09*. *The shoes* and *(on) the competition* are arguments to the predicates. Each sense has a number of predefined arguments for, e.g., the actor doing the deed, the object being manipulated, the source or the destination. These arguments are labelled as *A0*, *A1*, etc. Both the sense of the verbs and the arguments are determined using statistical methods described in [10].

The natural language service outputs a preliminary form of program statements derived from the sentences. However, the matching to actions and objects existing in the world is done in the Engineering System. In the simplest form a program statement contains an action (the predicate) and a few arguments (objects). The action is then mapped to a robot program template while the arguments are mapped to the physical objects in the workspace, using their names and types. Actions described this way can be picking, placing, moving and locating objects. More complicated program structures can be expressed using conditions that have to be maintained during the action or for stopping it, as in the sentence

	Assemble	the	shieldcan	to	the	PCB	using	shieldcaninsertion	.
<a href="#">assemble.02</a>		A1		A2		AM-ADV			
<a href="#">use.01</a>								A1	

Parsing sentence required 22ms.

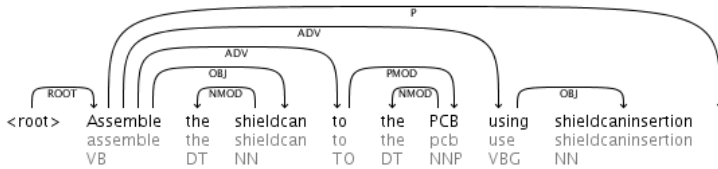


Figure 15: The result given by the parser of the sentence from Fig. 14. At the top, each line displays a found predicate with its arguments. Assemble was evaluated to assemble.02 with the arguments the shieldcan (A1), to the PCB (A2) and a manner using shieldcaninsertion. The bottom of the picture displays the dependency graph (actually a tree). The arrows point, beginning from the root of the sentence, from parents to children. Each arrow is labelled with the grammatical function of the child. Under each word the corresponding part-of-speech tag (determiner - DT; noun - NN, etc) can be found.

Search in the  $x$ -direction until contact while keeping 5 N in the  $z$ -direction. The example sentence *Assemble the shieldcan to the PCB using ShieldCanInsertion* given in Fig. 14 has a skill, *ShieldCanInsertion*, as argument to *use* (which in turn is a nested argument to *assemble*, see bottom of Fig. 15). *Use* is not mapped to a robot action, but rather prompts a search for a corresponding skill in the KIF libraries. The skill is instantiated with the arguments as parameters or, when no matching parameter can be found, with default values. For example, the *ShieldCanInsertion* is described in the ontology with an actuated object and a fixated object, which are mapped to A1 – *the shieldcan* and A2 – *the PCB*.

These programs can be further edited or directly executed on a physical robot or in the virtual environment of the Engineering System.

There exists also a scheduling service that helps the user to assign actions to a system with limited resources. The current implementation of the service is based on the list-scheduling. The manipulation skills require different end effectors, e.g., for gripping and for screwing. By adding a tool changer to the cell, the robot can change end effectors during the task. The time it takes to change tools is added as penalty on the priority of the actions. When there are multiple arms, one arm can of course change a tool while waiting for the other arm to finish its operation during a two-arm manipulation skill. A typical input to the service can be to schedule a partially ordered task on a two-armed robot with three tools and one force sensor. Each action lists its estimated time and the resource requirements, required tool(s) and resources. Given the estimated time to change tools and the number of cycles, the service will output a suggested schedule that minimizes the total time.

The last service named here is a code generation service used by the task execution system. It is described below in Section 7.

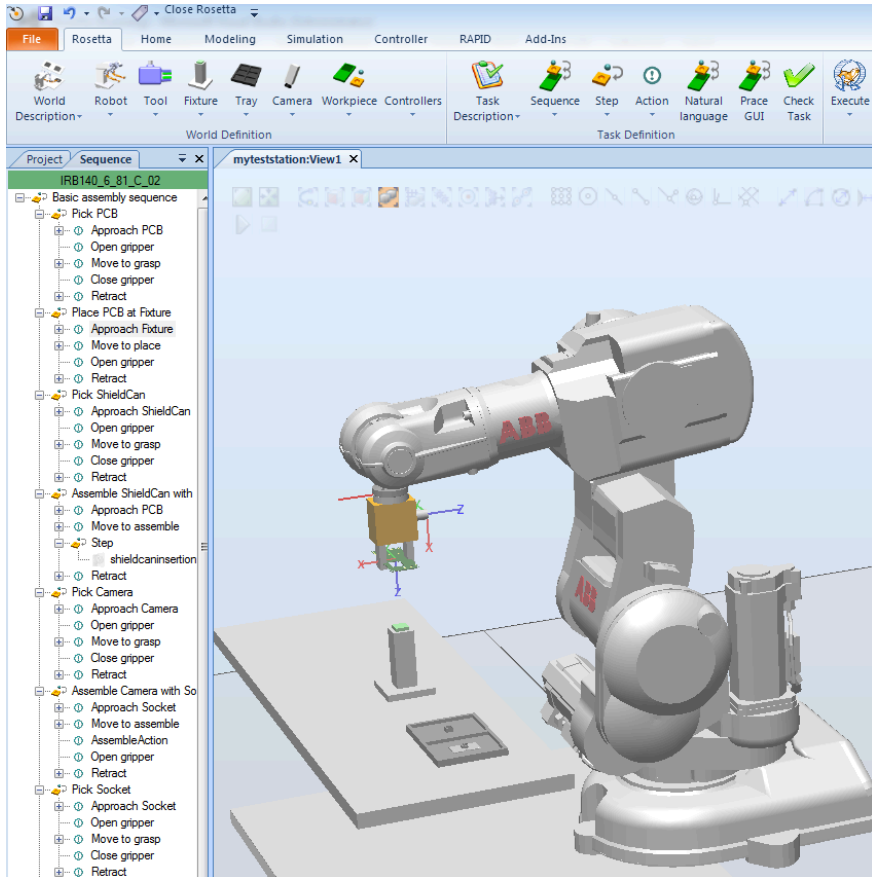


Figure 16: The engineering system is a plug-in the programming environment ABB RobotStudio.

## 6 Engineering System

The Engineering System is a high level programming interface implemented as a plugin to the programming and simulation IDE *ABB RobotStudio*,<sup>9</sup> shown in Fig. 16. When creating a station, objects such as the robot, workpieces, sensors, trays and fixtures can be manually generated in the station or downloaded from KIF together with the corresponding ontologies.

A physical object is characterized by its local coordinate frames, the *object frame* and a number of relative coordinate frames called the *feature frames*, see Fig. 17. Geometrical constraints are expressed as relations between feature frames, and may be visualized as in Fig. 18.

An example program sequence is shown in Fig. 19. The program has a nested hierarchy, where steps (such as pick or place) may contain atomic motions and gripper actions.

<sup>9</sup><http://new.abb.com/products/robotics/robotstudio>

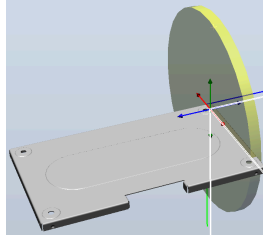


Figure 17: A feature frame.

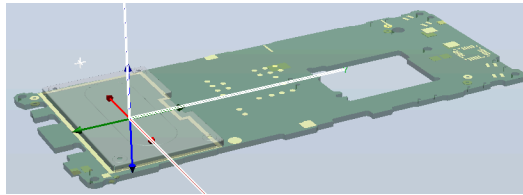


Figure 18: A geometrical relation between two objects.

## 7 Execution

The sequence from Fig. 19 is sent to the execution system, which in turn calls the code generation service that returns a complete state machine (serialized in an XML file), which is visualized, compiled and executed using JGrafchart tool [94]. It creates a task state machine, where each state is either a call to primitive functions on the robot, or a nested skill. Fig. 20 shows a small part of a generated state machine. Each skill is either retrieved from KIF and instantiated with the new parameters, or generated from scratch by creating a closed kinematic chain for a given robot and the objects. The vendor-specific code is executed using the native robot controller, while the more complex sensor-based skills are executed using an external control system [14] and the state machine switches between these two controllers when necessary.

To guarantee a safe execution, the injury risk for different velocities is evaluated using the data stored in KIF and the final robot speed is appropriately adjusted.

## 8 Related Work

Task representation has been an important area for the domain of robotics, in particular for autonomous robots research. The very first approaches were based on logic as a universal language for representation. A good overview of the early work can be found in [16]. The first autonomous robot, SHAKEY, exploited this approach to the extreme: its planning system STRIPS, its plan execution and monitoring system PLANEX and its learning component (Triangle tables) were all based on the first order logic and deduction [60]. This way of thought continued, leading to such efforts as "Naive physics" by Patrick Hayes (see [16]) or "Physics for Robots" [77]. This development stopped because of the insufficient computing power available at that time, but has recently received much attention in the wider context of semantic web. The planning techniques [30] have also advanced much and may be used

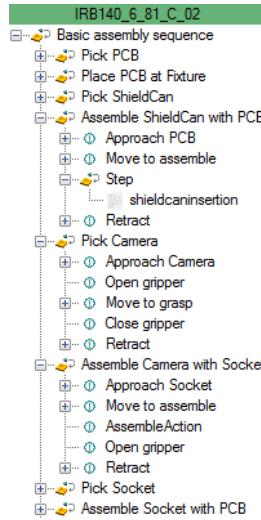


Figure 19: An example sequence of the cell phone assembly. First, the PCB is moved to the fixture. Then, the shieldcan is picked and inserted on the PCB using a sensor-based skill called shieldcaninsertion. The cell phone camera is assembled with the socket, and then the socket is inserted on the PCB.

nowadays for cases of substantial complexity, although generic automation problems are usually still beyond this limit.

Later, mixed architectures begun to emerge, with a reasoning layer on the top, reactive layer in the bottom, and some synchronization mechanism, realized in various disguises, in the middle. This approach to building autonomous robots is prevalent nowadays [6], where researchers try to find an appropriate interface between abstract, declarative description needed for any kind of reasoning, and procedural one needed for control. The problem remains open until today, only its complexity (or the complexity of solutions) grows with time and available computing power.

Task description in industrial robotics setting comes also in the form of hierarchical representation and control, but the languages used are much more limited (and thus more amenable to effective implementation). There exist a number of standardized approaches, based, e.g., on the IEC 61131 standards [36] devised for programmable logic controllers, or proprietary solutions provided by robot manufacturers, however, to a large extent the solutions are incompatible with each other. EU projects like RoSta<sup>10</sup> are attempts to change this situation.

At the theory level all the approaches combining continuous and discrete formalisms may be considered as variants or extensions of hybrid systems [31], possibly hierarchical. Hybrid control architectures allow us to some extent separation of concerns, where the continuous and real-time phenomena are handled in their part of the system, while the discrete aspects are treated by appropriate discrete tools. Our earlier work attempted at declaratively specifying such hybrid systems, but was limited to knowledge-based configuration [32].

Robotics systems are usually build from a number of distributed heterogenous hardware

<sup>10</sup>[www.robot-standards.org](http://www.robot-standards.org)

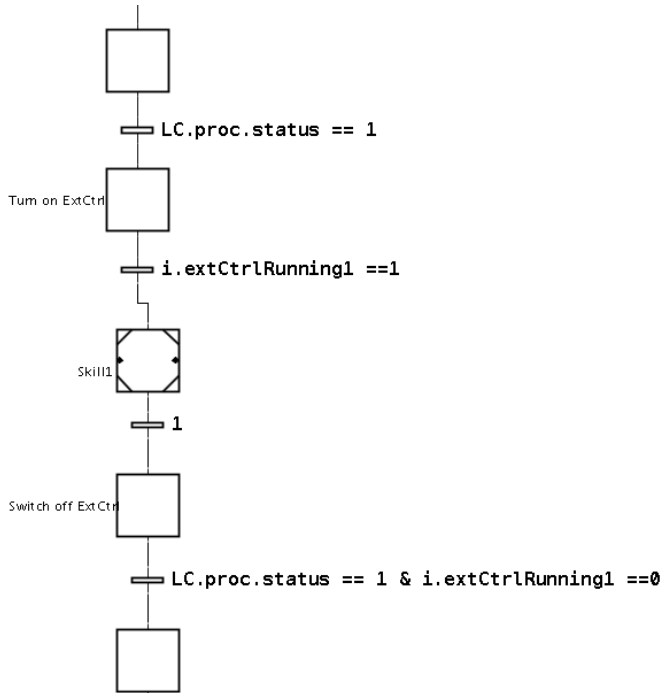


Figure 20: Each box is a state in the task state machine. The state called Skill 1 with marked corners is a nested state machine containing a (dynamically generated) sensor-based skill. Before and after the sensor-based skill the external controller is started and turned off, respectively.

and software components that have to seamlessly interact during execution phase. In order to simplify configuration, communication and hide the complexity of the system, as well as to promote portability and modularity, there exist several frameworks for robotics middleware (see comprehensive surveys [22] and [54]). Module functionality can be provided as nodes in the ROS 10 environment, or as standardized components in RT-components [61], where the modules can provide blackbox-type computations with well-specified interfaces.

Task descriptions come in different disguises, depending on the context, application domain, level of abstraction considered, tools available, etc. Usually tasks are composed out of skills, understood as capabilities of available devices [11], but the way of finding appropriate composition varies heavily, from manual sequencing in many workflows, via AI-influenced task planning [30], hybrid automata development tools [31], Statecharts [34] and Sequential Function Charts (SFCs) [36], iTaSC specifications [21], to development of monolithic programs in concrete robot programming languages, like, e.g., ABB RAPID.

There have been several attempts to codify and standardize the vocabulary of robotics. There exists an old ISO standard 8373 requiring however a major revision to suit the demands of contemporary robotics. IEEE Robotics and Automation Society is leading some work towards standardization of robotic ontologies. In particular, there are first drafts of robotic core ontology [19], although not as developed as the ROSETTA ontology described



in this paper. Regarding industrial robotics, the work on kitting ontologies, originated at NIST [4], may be considered as an early attempt to address the problem.

In the area of service robotics there are several systems exploiting the knowledge-based approach, and relying on an underlying ontology, like KNOWROB [89] (based on the generic OPENCYC ontology [51]), used in ROBOHOW project<sup>11</sup> or several participants in the ROBOEARTH project<sup>12</sup> [98]. However, they do not attempt to standardize the domain, as the variance of tasks and skills in the service robotics is very large. On the other hand, the KNOWROB ontology became a de-facto standard used in several experimental robot systems.

## 9 Conclusions

We have shown a generic knowledge-based system architecture and its possible use in industrial robotic systems. In particular, we have employed the approach for representing and realizing force-controlled tasks realized by one- and two-armed ABB robots in an industrial setting. The presented generic ontologies are either novel, or a derivative of our earlier research. The use of semantic tools and explicit knowledge in industrial robotics is in its early stage, with only a few other published examples [4]. The ideas have been experimentally verified and work well in the currently ongoing EU-projects PRACE<sup>13</sup> and SMErobotics.<sup>14</sup> The implemented system is just a proof of concept, and systems that are derived from this work must undergo usability, security and performance testing, before they might be considered to be ready for industrial practice. But already now it can be stressed that the knowledge-based approach allowed us to create composable representations of non-trivial assembly skills, shown to be reusable among different models of ABB robots, but also portable to other vendors and control architectures (like the one reported in [40] and running on a Kuka LWR4 robot).

The already ongoing continuation of the work presented above involves integration of a heterogeneous system consisting of a mobile robot platform (Rob@Work) running a ROS-based control system, and a real-time-enabled ABB-manipulator running the ABB-specific control software, so that the two parts can operate seamlessly together as an integrated, knowledge-based, productive robotic system. This work includes deploying knowledge-based services in the context of chosen robotic middleware.

Future work involves contribution to the IEEE standardization efforts, and aligning and sharing robotic ontologies with other research groups. An online documentation of the core ROSETTA ontology is also expected. The number of knowledge-based services should be extended with, e.g., online reasoning during execution, geometrical reasoning and integrated path planning and optimization. We are also verifying this approach in other domains of manufacturing, like wood-working and machining, expecting to extend the ontologies appropriately.

We have found out during the work described in this paper that skills are much more than just a potential to execute coordinated motions. This line of thought has been already present in [59], where business aspects of skills have been pointed to. We plan to explore this topic in the nearest future.

---

<sup>11</sup><http://robohow.eu>

<sup>12</sup><http://roboearth.org/>

<sup>13</sup><http://prace-fp7.eu/>

<sup>14</sup><http://www.smerobotics.org/>

## Acknowledgments

The research leading to these results has received partial funding from the European Union's seventh framework program (FP7/2007-2013) under Grant agreement nos. 230902 (project ROSETTA), 285380 (project PRACE) and 287787 (project SMErobotics).

The work described in this paper has been done in tight collaboration with many people from the project consortia. The authors are indebted for many fruitful discussions.

An early version of this paper has been presented during the 12th Scandinavian Conference on Artificial Intelligence, Aalborg, Denmark, October 2013.



# Paper II

---

## Natural Language Programming of Industrial Robots

Maj Stenmark

Pierre Nugues

Department of Computer Science  
Lund University  
maj.stenmark@cs.lth.se  
pierre.nugues@cs.lth.se

### ABSTRACT

In this paper, we introduce a method to use written natural language instructions to program assembly tasks for industrial robots. In our application, we used a state-of-the-art semantic and syntactic parser together with semantically rich world and skill descriptions to create high-level symbolic task sequences. From these sequences, we generated executable code for both virtual and physical robot systems. Our focus lays on the applicability of these methods in an industrial setting with real-time constraints.



# 1 Introduction

Robot programming is time consuming, complex, error-prone, and requires expertise both of the task and the platform. Within industrial robotics, there are numerous vendor-specific programming languages and tools, which require certain proficiency. However, to increase the level of automation in industry, as well as to extend the use of robots in other domains, such as service robotics and disaster management, it has to be possible for non-experts to instruct the robots.

Since humans communicate with natural language (NL), it is appealing to use speech or text as instruction means for robots as well. This is complicated for two main reasons: First, NL can be 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 parsed, 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, we introduce a method for using natural language to program robotized assembly tasks and we describe a prototype of it. 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. Such predicate-argument structures reflect common semantic situations described through language and at the same time use a logical representation. Using the predicate-argument structures, we can extract the orders embedded in a user's sentences and map them more easily onto robot instructions.

## 2 Related Work

Natural language programming for robots has been investigated for both service and navigational robots from the early 1970's. SHRLDU [99] is an oft-cited example of the first attempts to give robots conversational competences. To interpret and convert a user's sentences into instructions, robotic system often make use of an intermediate representation. Examples include [87, 47, 78], where the authors have developed their own domain specific semantic representation for navigational robots.

Tenorth et al. [88] parse pancake recipes in English from the World Wide Web and generate programs for their household robots. They use the WordNet lexical graph [69] with a constituent parser and they map WordNet's *synsets* to concepts in the Cyc [51] ontology. Finally, they add mappings to common household objects.

In order to bridge the sentence to the robot actions, all the examples mentioned above seem to use ad-hoc intermediate formalisms that are difficult to adapt to other domains, languages, or environments. Frame semantics [24] is an attempt to provide generic models of logical representations of sentences. Frame semantics starts from prototypical situations shared by a language community, English for instance, and abstracts them into frames. While frame semantics is only a theory, FrameNet [28, 76] is a comprehensive dictionary that provides a list of lexical models of the conceptual structures. Commercial situations like selling are represented with the Commerce\_sell predicate-argument structure, where the arguments include a buyer, a seller, and goods. Given a sentence and a verb belonging to this frame, like *vend*, *sell*, or *retail*, a semantic parser will identify the predicate and its arguments.

As of today, FrameNet has not a complete coverage of English verbs and nouns. Propbank [66] and Nombank [52] are subsequent projects related to FrameNet that both de-

veloped comprehensive databases of predicate-argument structures for respectively verbs and nouns and annotated large volumes of text with it. As training data is essential to the development of statistical semantic parsers, most of the current parsers use the Propbank nomenclature, as they are easier to train.

To the best of our knowledge, few robotics systems use existing predicate-argument nomenclatures. An exception is RoboFrameNet [92], a language-enabled robotic system that adopts frame semantics. However, the authors wrote their own frames inspired from FrameNet. Their model includes a decomposition of the frames into a sequence of primitives. They built a semantic parser that consists of a dependency parser and rules to map the grammatical functions to the arguments. Such techniques have been used from the early Absity system [35] and are known to have a limited coverage.

In the project, we describe below, we used a multilingual high-performance statistical semantic parser [12, 10] trained on the Penn Treebank and using the Propbank and Nombank lexicons. In contrast to RoboFrameNet, the parser we adopted can accept any kind of sentence.

### 3 System Overview

#### Architecture

The central part of the system architecture [11] is the *knowledge integration framework* (KIF). KIF consists of a client-server architecture where the server hosts ontologies, provides services, and object and skill libraries. The ontologies represent the world objects, such as robots, sensors, work-pieces and their properties, as well as robot skills. The skills are semantically annotated, platform-independent state machines, which are parameterized for reuse and executed using JGrafchart [94].

KIF interacts with the *engineering system* (ES), which is the high-level programming interface, and the robot controller. The ES is implemented as an extension to the programming and simulation environment ABB RobotStudio [1]. When creating the robot cell, the objects, such as sensors, work-pieces, and trays, can be generated or downloaded from KIF together with the ontology. Every physical object has an *object frame*, and a number of *feature frames* related to its object frame. These frames are used to express geometrical constraints; see Fig. 1.

A program consists of a sequence of steps, which in turn consists of actions, motions, skills, or nested steps. The sequence is created using the graphical interface of the ES. The steps for picking a printed circuit board (PCB) and placing it on a fixture are shown in Fig. 2. To execute the sequence, platform specific code (robot code or the XML file used by the state machine executor) is generated for the motions, actions and skills, and deployed on the target platform. To help the user quickly setup a skeleton sequence of a task, we provide a natural-language parsing service on KIF; see Fig. 3. The service reads the text input, parses the text in search of predicate-arguments structures, and returns those containing predicates that match the task vocabulary.

On the client side, the predicates are mapped to programs; the arguments representing station objects and the other parameters are filled with default values or geometrical relations taken from the station. The programmer can then check the sequence, possibly alter it, and finally execute it.

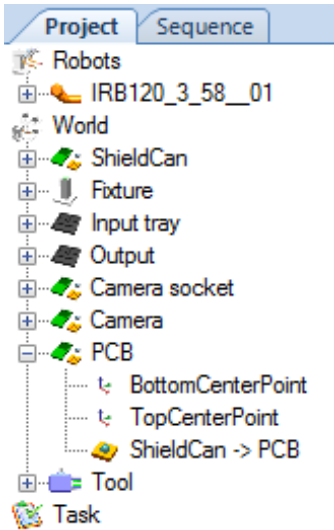


Figure 1: In the object browser, the robots are listed under robots; all physical objects are listed under world and each object lists its own frames and relations.

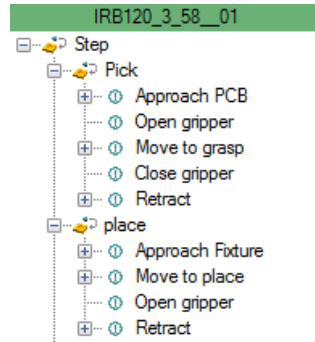


Figure 2: The visual rendering of a program for picking and placing a PCB.

### Predicate-Argument Structures

An assembly task can be defined as e.g.: *Pick the PCB from the input tray and place it on the fixture. Then take a shield can and insert it on the PCB.* These sentences are parsed to extract the predicates-argument structures *pick(PCB, input tray)* and *place(it, fixture)*, while the agent parameter, *robot*, is implicit.

The parser is trained on the Penn Treebank that uses the Propbank lexicon [37]. Propbank labels each English verb with a sense and defines a set of arguments that is specific to each verb. In the sentence: *Pick the PCB from the input tray and place it on the fixture*, both *pick* and *place* have sense 1 (*pick.01* and *place.01*):

- *Pick.01* has three possible arguments; *arg0*: agent, entity acquiring something, *arg1*: thing acquired and *arg2*: seller.
- *Place.01* has *arg0*: putter, *arg1*: thing put, and *arg2*: where put.

The parsing output is shown in Fig. 4. As shown in this figure, the *arg1* and *arg2* arguments to *pick.01* are matched to *the PCB* and *the input tray* respectively, while the robot (*arg0*) is implicit. Before mapping the identified arguments to the station objects, the arguments corresponding to the same entity have to be gathered into *coreference chains*; see Fig. 5. The last step links the coreference chains to the entities in the station using the object name or type.

### Task Vocabulary

The vocabulary is currently rather limited. We only considered predicates matching programs that the robot could generate. Each program has arbitrary language tags such as *take*, *insert*, *put*, *calibrate*, either predefined or edited by the user. Possible arguments to the programs are the objects in the station, which is a well-defined, finite world.



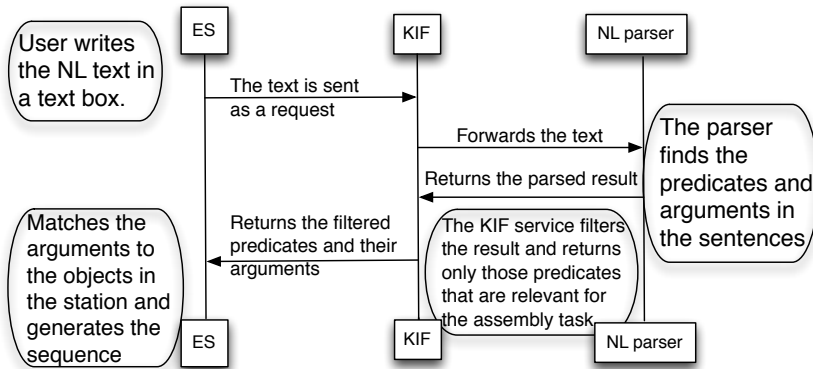


Figure 3: The data flow between the user, the KIF service and the semantic parser.

	Pick	the	PCB	from	the	input	tray	and	place	it	on	the	fixture	.
pick.01		A1	A2											
place.01										A1	A2			

Figure 4: Parsing result from the first sentence. The parser identified two predicates, pick and place, and two arguments for each predicate.

## 4 High-level Programming Prototype

On the highest level, the task is represented by an assembly graph [48], which is a partially ordered tree of assembly operations; see Fig. 6. The graph describes the assembly of an emergency stop button box.

Each operation specifies the desired geometrical relations of the involved objects and the skill type for the assembly. Examples of skill types in the ontology are *screw*, *glue* and *peg-in-hole*, where each type can have several different implementations. The assembly operations are subgoals, and the root node represents the final goal of the task. The motivation for the assembly graph is to have a platform independent task description, so that different implementations can be compared and reasoned about.

The assembly graph is realized by sequences of actions and motions for each robot. The sequence can be: 1) created manually by adding actions and motions one by one and editing their properties, 2) generated from the assembly graph or 3) created by using a natural language interface. An example of the latter is shown in Fig. 7: two assembly steps of a stop button box assembly are described by natural language. Fig. 8 shows the parsed result from Fig. 7.

Each predicate is mapped to a type of skill. For example, a *pick* or *take* consist of a sequence of primitive actions: approaching the object to be picked, opening the gripper, moving slowly to a grasp position, closing the gripper, and then retracting. The mapping of the objects are rudimentary: by name (ignoring space and case) or, if this is unsuccessful,

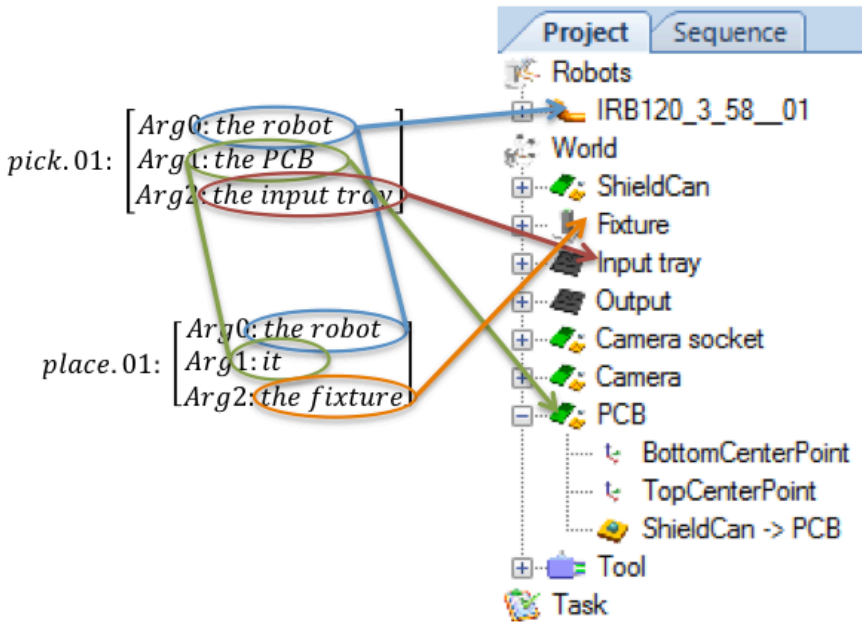


Figure 5: Coreference solving of entities in the first sentence. Mentions corresponding to the same entity are gathered into coreference chains.

by the ontology type (e.g. fixture, tray or pin). When generating the motions for picking and placing the objects, the application uses the existing grasp positions and relations between the work-pieces as default values. If no relations exist, a new one is created with zero offset. The actions for opening and closing the gripper are taken from the selected tool, since each tool describes its own procedures. The resulting sequence is shown in Fig. 9. Using reasoning services available from KIF, the generated sequence can then be checked for inconsistencies and additional skills are suggested to solve missing constraints (e.g. an object has to be placed in a fixture before an assembly or a tool needs to be exchanged between drilling and picking). The code generated from the sequence is executable on both virtual and physical robots; see Fig. 10. To expand the vocabulary, the user can add natural language tags to existing steps and upload them to KIF.

## 5 Conclusions

In this paper, we have presented a system to describe robot assembly tasks in the RobotStudio environment using natural language. From an input sentence, the processing pipeline applies a sequence of operations that parses the sentence and produces a set of predicate-argument structures. The semantic module uses statistical techniques to extract automatically these structures from the grammatical functions.

The NLP pipeline is designed so that it reaches high accuracies and has short response times required for user interaction. Parsing a sentence takes from 10 to 100 milliseconds.

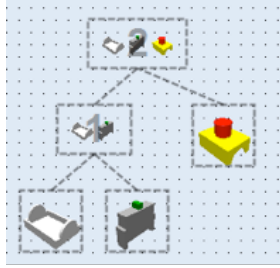


Figure 6: The assembly graph is created by dragging and dropping icons of the objects. Here, the first assembly operation involves the base of the emergency button (left) and the switch (right). In the second operation the lid is added to the subassembly.

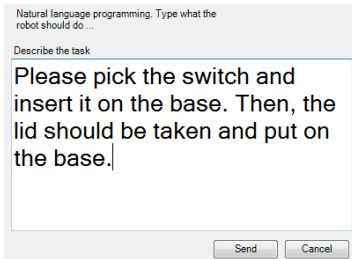


Figure 7: The commands are written into a simple text field, the narrative is then sent to the KIF service that facilitates semantic parsing.

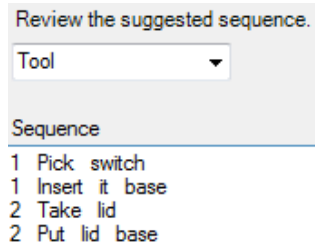


Figure 8: The result the parsed predicates along with their arguments.

Drawing from the frame semantics theory, the semantic parser uses a standardized inventory of structures and can be applied to unrestricted text. This makes the pipeline more easily adaptable to new tasks and new environments.

As second step, the system maps the predicate and the arguments extracted from the sentence to robot actions and objects of the simulated world. These objects and actions are stored in a unified architecture, the *knowledge integration framework* that represents and manages the entities, services, and skill libraries accessible to the robot.

Making the application part of a tool already used by industry is a conscious choice: high-level natural language programming is convenient to get an application up and running quickly. However, when tuning the parameters of a task, the programmer can still use the traditional tools, e.g. to edit the generated code directly. Also, because of the industrial focus, we have real-time performance on the underlying sensor and control systems, which is necessary for many manipulation tasks in assembly operations.

Unlike previously reported results, our approach supports both a command-like interface and parsing of longer texts, yielding multistep programs.

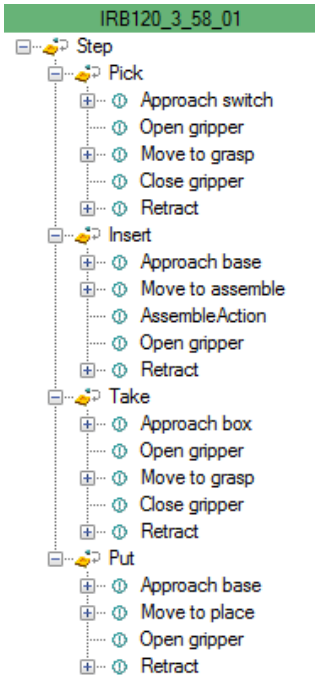


Figure 9: The generated sequence for inserting a switch on the base of a stop bottom and putting the top of the box on the base.



Figure 10: The sequence from Fig. 8 executed on a physical robot.

## 6 Future Work

The obvious drawback of this implementation is the lack of speech as an input modality. However, since many smartphones have sufficient speech recognition for our purposes, this was not our main scientific concern. Rather, we wanted to extend the skill library with relevant and generic assembly skills. We plan to extend our application with tools that make it simple to extract the natural language predicate-argument structures given a skill, its parameters (objects, velocities, forces), and a textual description of the skill. Another extension is to automatically search after suitable implementations that are tagged with synonyms to the used words.

## 7 Acknowledgments

The research leading to these results has received funding from the European Union's seventh framework program (FP7/2007-2013) under grant agreements N° 230902 (ROSETTA) and N° 285380 (PRACE) and from the Swedish Research Council grant N° 2010-4800 (SEMANTICA).



# Paper III

---

## Paper III: Describing Constraint-Based Assembly Tasks in Unstructured Natural Language

Maj Stenmark

Jacek Malec

Department of Computer Science  
Lund University  
maj.stenmark@cs.lth.se  
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 [83] 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.



# 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 [83] 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 [7].

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 [33]. 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 [36]. Using a touch screen as an input device, icon-based programming languages such as in [9] can also lower the threshold to robot programming.

Natural language programming for robots has been investigated since the early 1970's. SHRLDU [99] is an example of the first attempts to give robots conversational competences. To interpret and convert a user's sentences into instructions, robotic system often make use of an intermediate representation. Examples include [47] and [87], where the authors have developed their own domain-specific semantic representations for robot navigation.

[88] parse pancake recipes in English from the World Wide Web and generate programs for their household robots. They use the WordNet lexical database [69] with a constituent parser and they map entries in the WordNet dictionary to concepts in the Cyc ontology [51].



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 [76], based on frame semantics, is a comprehensive dictionary that provides a list of lexical models of the conceptual structures. Propbank [66] has developed an 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 [92]. 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 [12, 10] 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 [80]. 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 [83, 80]; 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 [1] visual IDE.

**Objects in the World** The core ontology, *rosetta.owl* [80], contains devices such as

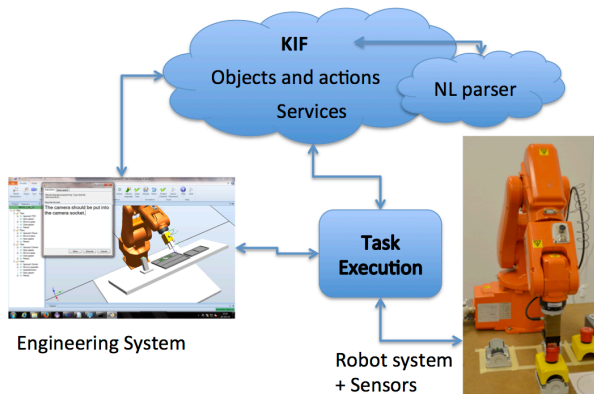


Figure 1: A view of the system architecture.

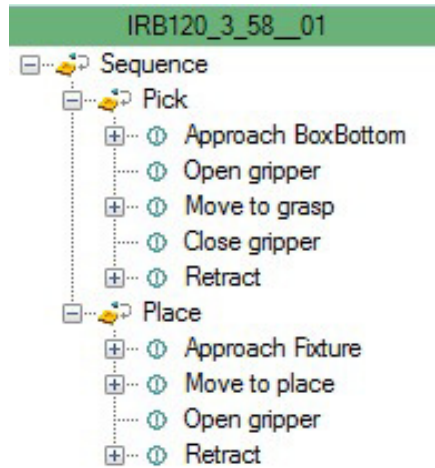


Figure 2: A sequence of skills.

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 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 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<sup>15</sup> [10] that outputs predicate-argument structures

<sup>15</sup>The parser is available as open source software, freely accessible at <http://barbar.cs.lth.se:8081/>.

in standard format (cf. Fig. 3).

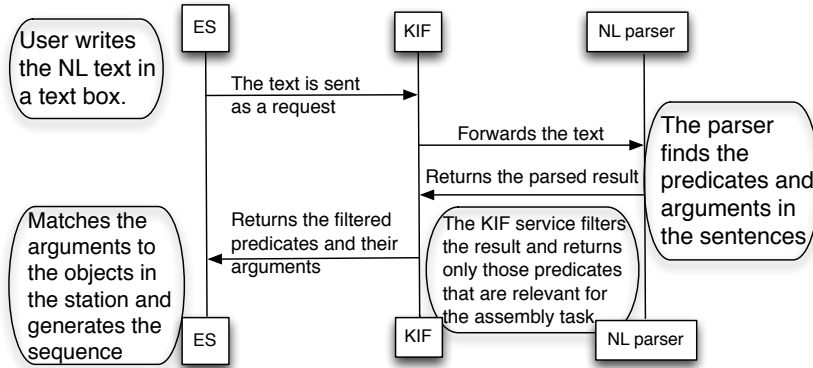


Figure 3: The NL parsing sequence.

**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 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.

Previous work [83] 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, parallel 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 [21], together with external force/torque sensors. These skills are specified by state machines using Grafchart [94] language, where states are simple motions and transition conditions are, e.g., timeouts or force and torque thresholds. A motion is specified by

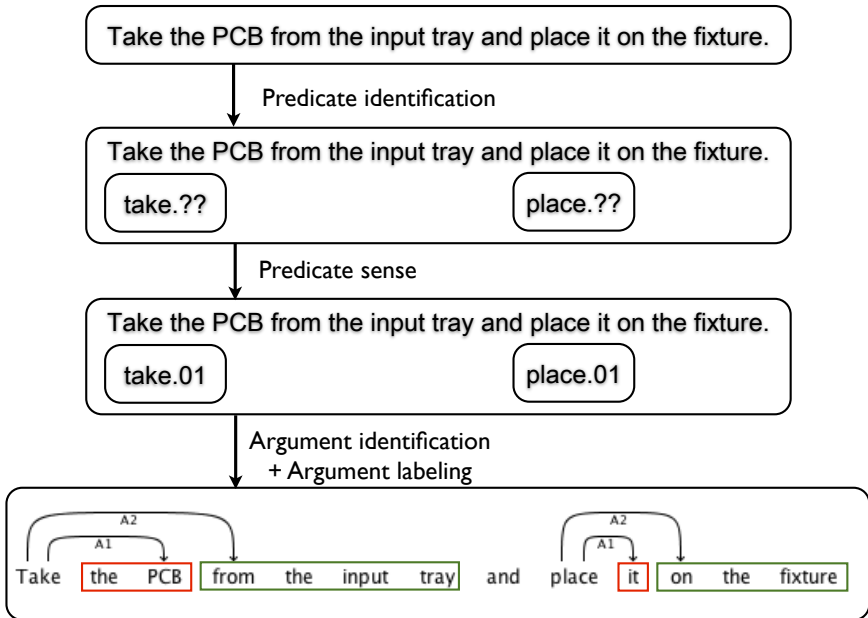


Figure 4: The parsing pipeline.

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 [84].

## 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

**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
    end

```

---

**return** *actions* and *unknownStatements*

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 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.

---

**Function** sortPredicates(*semOutput* put)

---

```

if semOutput has a root element then
  | Let q be an empty queue
  | root ← 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 ← all predicates from semOutput in input order
  | Add all predicates to q
end
return q

```

---



---

**Function** createArgs(*p*)

---

```

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

```

---

---

**Function** findArgs(*p*)

---

```
a1 ← argument "A1" of p
if a1 does not exist then
  | a1 ← search for an argument labelled "TMP", "IN", "AM-LOC"
  | if a1 is not found then
  | | a1 ← search among children to p labelled "LOC"
  | end
end
a2 ← argument "A2" in p
if a2 is not found then
  | a2 ← search for an argument labelled "TMP", "IN", "AM-LOC"
end
if a1 is not found and a2 is found then
  | a1 ← a2
  | a2 ← void
end
return (a1, a2)
```

---

---

**Function** recursiveSearch(*w*)

---

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

---

---

**Function** getWildcard(*p*)

---

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

---

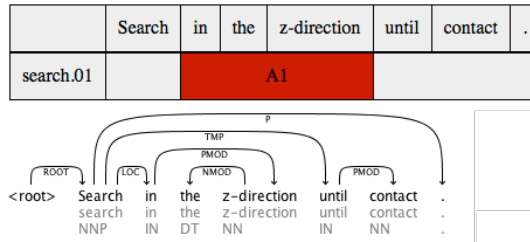


Figure 5: The parse result of “Search in the z-direction until contact”, together with the dependency graph.

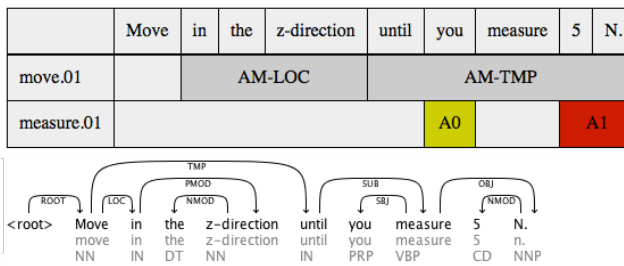


Figure 6: The parse result of “Move in the z-direction until you measure 5 N”.

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

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 parallel actions of searching and holding, while the search is a nested *until*-statement with the transition condition.



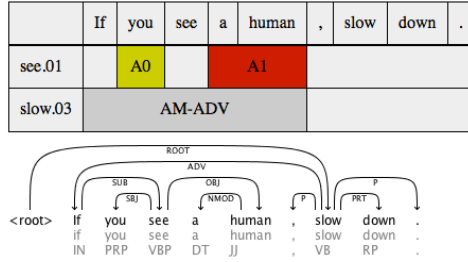


Figure 7: Result for an if-sentence.

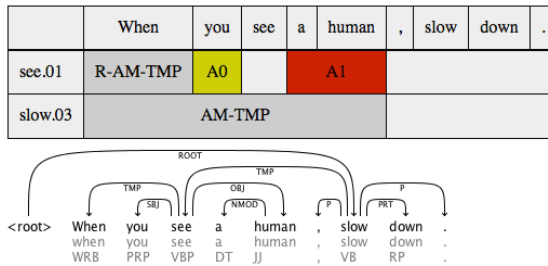


Figure 8: Result for a when-sentence.

**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.

**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.

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 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*.

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

Figure 9: Example of the usage of the wildcard word use.

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

Figure 10: Example of the usage of the word with.

**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 parallel 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 [85].

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.



# Paper IV

---

## From High-Level Task Descriptions to Executable Robot Code

Maj Stenmark

Jacek Malec

Andreas Stolt

1. Department of Computer Science
  2. Department of Automatic Control
- Lund University  
maj.stenmark@cs.lth.se  
jacek.malec@cs.lth.se  
andreas.stolt@control.lth.se

### ABSTRACT

For robots to be productive co-workers in the manufacturing industry, it is necessary that their human colleagues can interact with them and instruct them in a simple manner. The goal of our research is to lower the threshold for humans to instruct manipulation tasks, especially sensor-controlled assembly. In our previous work we have presented tools for high-level task instruction, while in this paper we present how these symbolic descriptions of object manipulation are translated into executable code for our hybrid industrial robot controllers.



# 1 Introduction

Deployment of a robot-based manufacturing system involves a substantial amount of programming work, requiring background knowledge and experience about the application domain as well as advanced programming skills. To set up even a straightforward assembly system often demands many days of work of skilled system integrators.

Introducing sensor-based skills, like positioning based on visual information or force-feedback-based movements, adds yet another level of complexity to this problem. Lack of appropriate models and necessity to adapt to complexity of the real world multiplies the time needed to program a robotic task involving continuous sensor feedback. The standard robot programming environments available on the market do not normally provide sufficient sensing simulation facility together with the code development for specific industrial applications. There are some generic robot simulators used in research context that allow simulating various complex sensors like lidars, sonars or cameras, but the leap from such simulation to an executable robot code is still very long and not appropriately supported by robot programming tools.

The goal of our research is to provide an environment for robot task programming which would be easy and natural to use, even for plain users. If possible, that would allow simulation and visualization of the programmed task before the deployment phase, and that would offer code generation for a number of predefined robot control system architectures. We aim in particular at ROS-based systems and ABB industrial manipulators, but also other systems are considered.

In our work we have developed a system for translation from a high-level, task-oriented language into either the robot native code, or calls at the level of a common API like, e.g., ROS, or both, and capable to handle complex, sensor-based actions, likewise the usual movement primitives.

This paper focuses on the code generation aspect of this solution, while our earlier publications described the task-level programming process in much more detail [11, 48, 80, 81].

Below we begin by describing the system architecture and the involved, already existing components. Then we proceed to the presentation of the actual contribution, namely the code generation process. In the next section we describe the experiments that have been performed in order to validate this approach. Finally we present a number of related works. The paper ends with conclusions and suggestions for future work.

## 2 System Overview

The principles of knowledge-based task synthesis developed earlier by our group [11, 13] may be considered in light of the Model-Driven Engineering principles [39]. In particular, the system described in the rest of this paper realizes the principles of separation of concerns, and separation of user roles, as spelled out recently

in robotic context in [95]. It consists of the following components:

- An intuitive task-definition tool that allows the user to specify the task using graphical menus and downloading assembly skills from a knowledge base, or by using a natural-language interface [81, 83];
- An advanced graphical simulation and visualization tool for ABB robots, extended with additional capabilities taking care of other hardware used in our experiments;
- Software services transforming the task specification into a combination of a transition system (a sequential function chart) and low level code executable natively on the robot controller;
- Controllers specific for the hardware used: IRC5 and custom ExtCtrl [14] for the ABB industrial robots, and ROS-based<sup>16</sup> for the Rob@Work mobile platform;
- ABB robots: a dual-arm concept robot, IRB120 and IRB140, Rob@Work platform from Fraunhofer IPA<sup>17</sup>, Force/Torque sensors from ATI Industrial Automation<sup>18</sup> used in the experiments mentioned in this paper, as well as vision sensors (Kinect and Raspberry Pi cameras) used for localization.

The functional dependencies in the system are illustrated in Fig. 1. The knowledge base, called Knowledge Integration Framework (KIF), is a server containing robotic ontologies, data repositories and reasoning services, all three supporting the task definition functionality [13, 48, 80]. It is realized as an OpenRDF Sesame (<http://www.openrdf.org>) triple store running on an Apache Tomcat servlet container (<http://tomcat.apache.org>). The Engineering System

<sup>16</sup>[www.ros.org](http://www.ros.org)

<sup>17</sup><http://www.care-o-bot.de/en/rob-work.html>

<sup>18</sup><http://www.ati-ia.com>

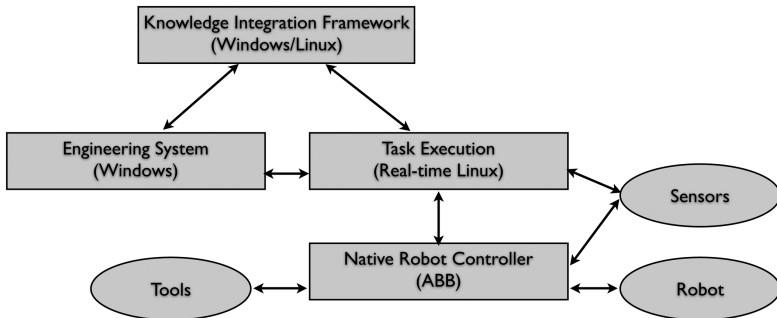


Figure 1: The Knowledge Integration Framework provides services to the Engineering System and the Task Execution. The latter two communicate during deployment and execution of tasks. See also Fig. 5.

(ABB RobotStudio [1]) is a graphical user interface for high-level robot instruction that uses the data and services provided by KIF for user support. The Engineering System uses the ontologies provided by KIF to model the workspace objects and downloads known skills and tasks from the skill libraries. Similarly, new objects and skills can be added to the knowledge base via the Engineering System. Skills that are created using classical programming tools, such as various state machine editors (like, e.g., JGrafchart [90], used both as a sequential function chart [36]—a variant of Statecharts [34]—editor, and its execution environment), can be parsed, automatically or manually annotated with semantic data, and stored in the skill libraries.

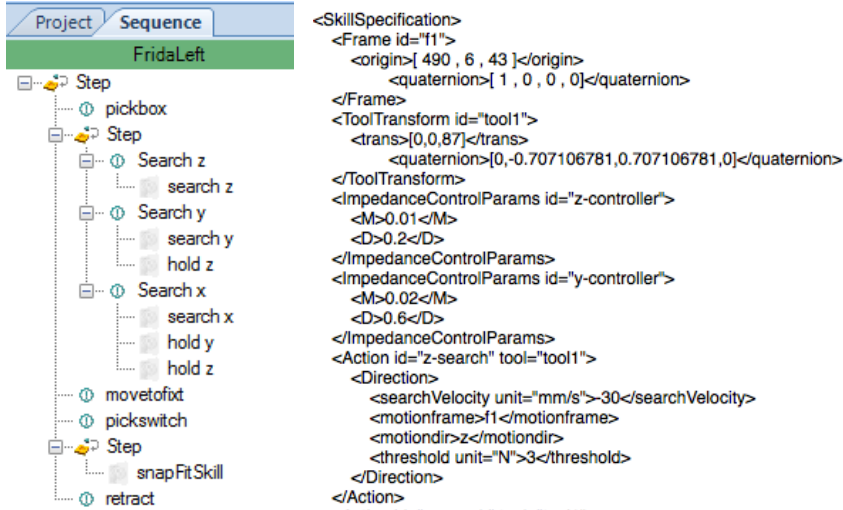
The Task Execution module is built on top of the native robot controller and sensor hardware. It compiles, with the help of KIF, a symbolic task specification (like the one shown in Fig. 2) into generic executable files and, when needed, hardware-specific code, before executing it. It is implemented on a real-time-enabled Linux machine, linking the external control coming from JGrafchart (a simple example is shown in Fig. 2b) or possibly other software, with the native controller of the robot. Depending on the system state (execution or teaching mode) or the action being carried out, the control is switched between the `ExtCtrl` system for sensor control and the native controller, allowing smooth integration of the low-level robot code with the high-level instructions expressed using the SFC formalism. It also runs adaption and error detection algorithms. The native robot controller is in our case an ABB IRC5 system running code written in the language RAPID, but any (accessible) robot controller might be used here. The Engineering System uses among other tools a sensor-based-motion compiler [84] translating a symbolic, constraint-based [21] motion specification into an appropriately parametrized corresponding SFC and the native controller code.

In addition to the benefit of providing modular exchangeable components, the rationale behind KIF as a separate entity is that the knowledge-providing services can be treated as black boxes. Robot and system-integration vendors can offer their customers computationally expensive or data-heavy cloud-based services [82] instead of deploying them on every site and each installation.

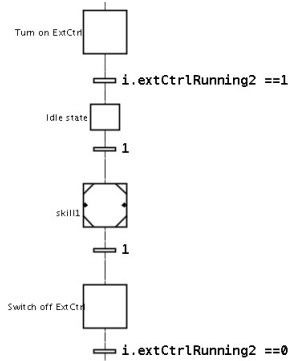
### 3 Code Generation

In order to illustrate the process of code generation, we will use an example task where a switch is assembled into the bottom of an emergency stop box. Both parts are displayed in Fig. 3a. The task is described in the Engineering System as a sequence, shown earlier in Fig. 2a. First the box is picked and aligned to a fixture with a force sensor. Then the switch is picked and assembled with the box using a snap-fit skill. The sequence is mixing actions (`pickbox`, `movetofixt`, `pickswitch` and `retract`) that are written in native robot controller code (ordinary blind moves), guarded search motions which are actions that are force-controlled (alignment to the fixture), and it also reuses a sensor-based skill (`snapFitSkill`). In this section we present how we generate and execute





(a) The task is shown as a sequence in Engineering System.



(b) A small part of the state chart generated from the sequence in Fig. 2a.

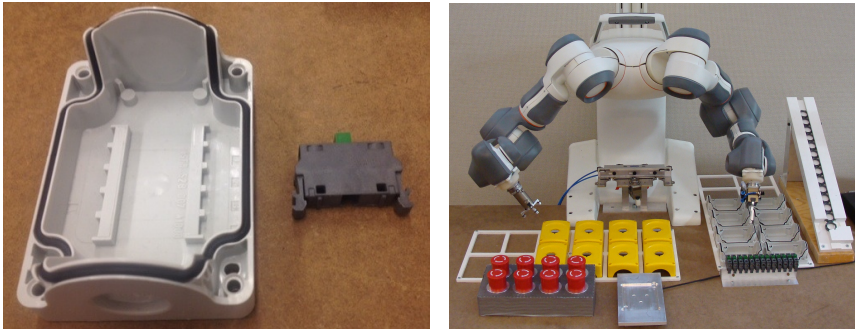
```

<SkillSpecification>
  <Frame id="f1">
    <origin>[ 490 , 6 , 43 ]</origin>
    <quaternion>[ 1 , 0 , 0 , 0 ]</quaternion>
  </Frame>
  <ToolTransform id="tool1">
    <trans>[0,0,87]</trans>
    <quaternion>[0,-0.707106781,0.707106781,0]</quaternion>
  </ToolTransform>
  <ImpedanceControlParams id="z-controller">
    <M>0.01</M>
    <D>0.2</D>
  </ImpedanceControlParams>
  <ImpedanceControlParams id="y-controller">
    <M>0.02</M>
    <D>0.6</D>
  </ImpedanceControlParams>
  <Action id="z-search" tool="tool1">
    <Direction>
      <searchVelocity unit="mm/s">30</searchVelocity>
      <motionframe>f1</motionframe>
      <motiondir>z</motiondir>
      <threshold unit="N">3</threshold>
    </Direction>
  </Action>
  <Action id="y-search" tool="tool1">
    <Direction>
      <searchVelocity unit="mm/s">40</searchVelocity>
      <motionframe>f1</motionframe>
      <motiondir>y</motiondir>
      <threshold unit="N">3</threshold>
    </Direction>
    <Constraint>
      <type>forcecontrolled</type>
      <controllerId>z-controller</controllerId>
      <motionframe>f1</motionframe>
      <motiondir>z</motiondir>
      <value unit="N">3</value>
    </Constraint>
  </Action>

```

(c) A sample XML description corresponding to the guarded motion skill from Fig. 2a that is sent to the code generation service by Engineering System. The parameter values are either set automatically or by the user in the Engineering System. If a guarded motion is generated, e.g., from text and one of the parameters is an impedance controller, the controller is selected among the controller objects in the station. All mandatory parameters must be specified before the code generation step.

Figure 2: A task can be created using the graphical interface of the Engineering System or by services for automatic sequence generation. The sequence shown is part of an assembly of an emergency stop button (see next section), consisting of a synthesized guarded motion, a complex snapFitSkill and three position-based primitives, see Fig. 2a. In Fig. 2b the step named skill1 is a macro step containing the synthesized guarded motion skill. Before and after the actual skill the steps for starting and turning off ExtCtrl are inserted. The idle state resets all reference values of the controller. Finally, Fig. 2c presents the corresponding input to the code generation service.



(a) The parts that are used in the process: the bottom of an emergency stop box (later "box") and a switch that will be inserted into the box. (b) The two-armed ABB robot and the workspace setup.

Figure 3: The example setup for the assembly experiments.

code for tasks containing these three types of actions. As an example we will use the sequence shown in Fig. 2a that, when executed, requires switching between the native robot controller and the external, sensor-based control (`ExtCtrl`).

The task sequence is translated into executable code in two steps. First, the native code for each primitive action is deployed on the robot controller. In this case RAPID procedures and data declarations are added to the main module and synchronized to the ABB controller from the Engineering System. In the second step a KIF service generates the task state machine (encoded as an SFC). Thus, KIF acts both as a service provider and a database, where the service builds a complete SFC, which can include steps synthesized from skills that are stored in the KIF databases. The final SFC is executed in JGrafchart, which, when necessary, calls the RAPID procedures on the native controller. The data flow between the modules is illustrated in Fig. 4.

### 3.1 Execution System Architecture

The execution system architecture is depicted in Fig. 5. The task is executed in JGrafchart, which in turn invokes functions on different controllers. The external controller (`ExtCtrl`) is implemented using Matlab/Simulink Real Time Workshop. It sends position and velocity references to the robot while measurements from the sensors are used to control the motion. Motions are specified using a symbolic framework based on iTaSC [21], by constraining variables such as positions, velocities or forces in a closed *kinematic chain* that also contains the robot. The communication between the modules is done using a two-way protocol called LabComm (<http://wiki.cs.lth.se/moin/LabComm>). LabComm packages data in self-describing samples and the encoders and decoders may be generated for multiple languages (C, Java, RAPID, C#). The `ExtCtrl` interface divides the

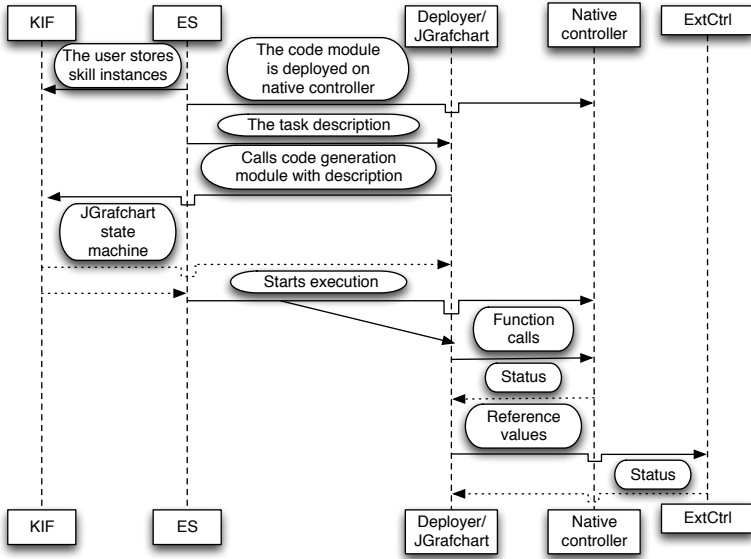


Figure 4: The Engineering System (ES) sends the task description to a small helper program called Deployer which in turn calls the code generation service on KIF, loads the returned file and starts JGrafchart.

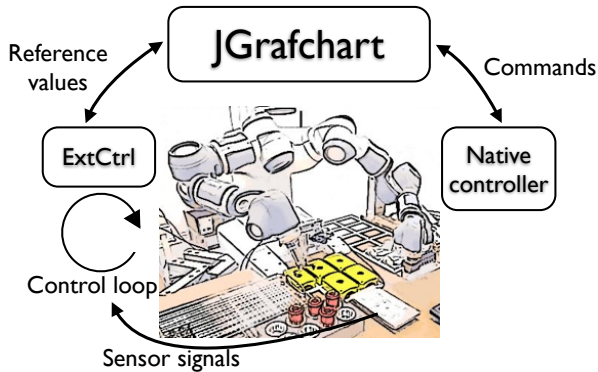


Figure 5: A schematic image of the execution architecture. The task state machine is executed in JGrafchart, which in turn sets and reads reference values to ExtCtrl and sends commands to the native controller.

samples into four categories: inputs, outputs, parameters and log signals. Hence, JGrafchart can set output signals and read inputs from the underlying controller.

LabComm is also used to send commands (strings and acknowledgements) to the native controller. In that sense, the protocol aligns well with ROS messages,

and two-way LabComm-ROS bridges have also been created. This is important since a few of our robot systems are ROS-hybrids, where an ABB manipulator is mounted on top of a ROS-based mobile platform, each having a separate LabComm channel to JGrafchart.

### 3.2 Sequential function charts in JGrafchart

JGrafchart is a tool for graphical editing and execution of state charts [90]. JGrafchart is used for programming sensor-based skills and has a hierarchical structure where state machines can be nested. For each robot, the generated state machine will be a sequence. Each primitive or sensor-based skill is represented by a state (step), and transitions are triggered when the primitive action or skill has finished. Each state can either contain a few simple commands or be a nested state machine, put into a so called macro step (in Fig. 2b shown by a square with marked corners). The generated and reused skills are put into these macro steps while primitive actions becomes simple steps with function calls.

When alternating between sensor-based external control and the native controller, the controllers are turned on and off during the execution, so these steps need to be added as well during the generation phase. The switching between controllers is handled by the state machine in JGrafchart. When `ExtCtrl` is turned on or off, the robot has to stand still to avoid inconsistent position and velocity values. When a controller is turned on it starts by updating its position, velocity and acceleration values to the current values on the robot.

The state machine can have parallel activities and multiple communication channels at the same time. Hence, code can be generated for multiple tasks and executed in parallel. Although the state machine allows synchronization between the tasks, we do not have a high-level representation of synchronized motions yet.

Finally, the sequence IDs and graphical elements, such as positions of the blocks, have to be added in order to provide an editable view. We generate very simple layout, however, much more could be done with respect to the legibility of the generated SFCs.

### 3.3 Code generation service

The code generation is implemented as an online service which is called by the Engineering System. It takes an XML description with the sequence as input and outputs the XML-encoding of the sequential function chart understood by JGrafchart. An example of the input is shown in Fig. 2c. Each robot has its own task, which needs to specify what LabComm port it will connect to. A primitive is specified by its procedure name and parameters to the procedure. Reusable skills are referenced by their URI, which is the unique identifier that is stored in the KIF repositories.

### 3.4 Reusing skills

A skill that is created in JGrafchart as a macro step, can be uploaded to KIF and reused. During the upload, it is translated into RDF triples. The skills are annotated with types, e.g., SnapFit, and skill parameters that are exposed to the users are also annotated with types and descriptions. The RDF representation is a simple transformation, where each state in the state machine is an RDF node annotated as a *State*, together with parameters belonging to the state, the commands, a description of the state (e.g. *Search x*) and is linked to transitions (which similarly are annotated with type and values). In this way, the parameters can be retrieved and updated externally using the graphical view in the engineering system. When a skill is updated in the engineering system, the new instance is also stored in KIF with the new parameter values. The URI in the input XML file refers to the updated skill, that is retrieved during the code generation process and translated back from triples to XML describing a macro step. The macro step is then parameterized and added as a step in the task sequence XML.

### 3.5 Guarded motions

One drawback of using the reusable skills is that there are implicit assumptions of the robot kinematics built into them, and thus the skill can only be used for the same (type of) robot. This limitation can be avoided by using a symbolic skill description and regenerating the code for each specific robot. This is what we do for the guarded motions. In this case, the skill specification is larger, as shown in Fig. 2c, where three actions are described. First, a search in the negative z-direction of the force sensor frame (f1) is performed. When the surface is hit, the motion continues in negative y-direction of the same frame while holding 3 N in the z-direction, pushing the piece to the side of the sensor. The last motion is in the x-direction while both pressing down and to the side, until the piece is lodged into the corner. In order to setup the kinematic chain, the coordinate frames that are used to express the motions have to be set, as well as the tool transform, that is, the transformation from the point where the tool is attached on the robot flange to the tip of the tool. Each constraint is specified along an axis of a chosen frame. There can be one motion constraint (using the <Direction> tag) which specifies the motion direction, speed and the threshold value for stopping. The other rotational and translational axes can also be constrained. The constraint should also specify what set of impedance controller parameters to use. Knowing what robot the code is generated for, the control parameters for the kinematic chain are set to the values of the frames and each motion sets reference values on corresponding parameters. Simply put, it is a mapping, where several hundred output signals have to get a value, where most are just dependent on the robot type, while some represent the coordinates of the frames in the kinematic chain and other reference values during execution. During the code generation the right value has to be set to the corresponding reference output signal and this is calculated depending on what frame is used.

Zone	z50
Speed	v200
Reference object	fixture
Actuating object	BoxBottom
	<div style="border: 1px solid black; padding: 2px;"> None  switch  BoxBottom  BoxTop  Tray_graphics  E-Button  Tool </div>
<b>Actuating object</b>	Tool

Figure 6: The properties of a move primitive: zone data for specifying maximal allowed deviation from the target point, velocity in mm/s, the position(s) of the motion specified by a relative position of the actuated object to a (frame of a) reference object. A motion can have a list of positions added to it.

### 3.6 RAPID code generation

The actions that have native controller code are called primitives. There are several different primitives and, in fact, they do not have to be simple. The most used are simple linear motions, move primitives for translation and rotation, and actions for opening and closing the gripper. The gripper primitives are downloaded together with the tool. The simplest form of a primitive is pure native code, a RAPID primitive, which does not have any semantically described parameters but where the user can add arbitrary lines of code which will be called as a function in the program. This is an exception though, since most primitives are specified by their parameters. E.g., the properties of a linear move are shown in Fig. 6. The target positions will be calculated from the objects' CAD-models and the objects' relative frames and positions in the virtual environment. The code for each primitive type and target values are synchronized to the controller as RAPID procedures and data declarations.

Hence, JGrafchart will invoke a primitive function with a string consisting of the procedure name followed by comma-separated parameters, e.g. "MoveL target\_1, v1000, z50". The string value of the procedure name can be invoked directly with late binding, however, due to the execution model of the native controller the optional parameters have to be translated into corresponding data types, the target name must be mapped to a robtarget data object and, e.g., the speed data has to be parsed using native functions.

## 4 Experiments

In order to verify that the code generation works as expected, we tested it using the sequence from the Engineering System depicted in Fig. 2a which resulted in an executable state machine, the same that is partly shown in Fig. 2b. The state machine is the nominal task execution, without any task-level error handling procedures. We have generated code for a two-armed ABB concept robot (see Fig. 3b) and the generation for guarded motions is working for both the left and the right arm, as well as for ABB IRB120 and IRB140 manipulators.

## 5 Related Work

The complexity of robot programming is a commonly discussed problem [67, 75]. By abstracting away the underlying details of the system, high-level programming can make robot instruction accessible to non-expert users. However, the workload for the experienced programmer can also be reduced by automatic generation of low-level control. Service robotics and industrial robotics have taken somewhat different but not completely orthogonal paths regarding high-level programming interfaces. In service robotics, where the users are inexperienced and the robot systems are uniform with integrated sensors and software, programming by demonstration and automatic skill extraction is popular. A survey of programming-by-demonstration models is presented by Billard [7].

Task description in industrial robotics setting comes also in the form of hierarchical representation and control, but the languages used are much more limited (and thus more amenable to effective implementation). There exist a number of standardized approaches, based e.g., on IEC 61131 standards [36] devised for programmable logic controllers, or proprietary solutions provided by robot manufacturers, however, to a large extent incompatible with each other. EU projects like RoSta [58] ([www.robot-standards.org](http://www.robot-standards.org)) are attempting to change this situation.

In industrial robotics, programming and demonstration techniques are used to record trajectories and target positions e.g., for painting or grinding robots. However, it is desirable to minimize downtime for the robot, therefore, much programming and simulation is done offline whereas only the fine tuning is done online [53, 15, 33]. This has resulted in a plethora of tools for robot programming, where several of them attempt to make the programming simpler, e.g., by using visual programming languages. The graphics can give meaning and overview, while still allowing a more advanced user to modify details, such as tolerances. In robotics, standardized graphical programming languages include Ladder Diagrams, Function Block Diagrams and Sequential Function Charts. Other well known languages are LabView, UML, MATLAB/Simulink and RCX. Using a touch screen as input device, icon-based programming languages such as in [9] can also lower the threshold to robot programming. There are also experimental systems using human programmer's gestures as a tool for pointing the intended robot locations [57]. However, all the systems named above offer monolithic compilation to the native code of the robot controller. Besides, all the attempts are done at the level of robot motions, focusing on determining locations. Experiences show [86] that even relatively simple sensor-based tasks, extending beyond the "drag and drop" visual programming using those tools, require a lot of time and expertise for proper implementation in mixed architecture like ours.

Reusable skill or manipulation primitives are a convenient way of hiding the detailed control structures [43]. The approach closest to ours is presented in the works of M. Beetz and his group, where high-level actions are translated, using knowledge-based techniques, into robot programs [5]. However, the resulting code is normally at the level of ROS primitives, acceptable in case of service robots, but without providing any real-time guarantees needed in industrial setting. In this

context, they also present an approach to map high-level constraints to control parameters in order to flip a pancake [42].

## 6 Conclusions and Future Work

In this paper we have described how we generate executable code for real-time sensor-based control from symbolic task descriptions. Previous work in code generation is limited to position-based approaches. The challenge to go from high-level instructions to robust executable low-level code is an open-ended research problem, and we wanted to share our approach in high technical detail. Naturally, different levels of abstraction have different power of expression. Thus, generating code for different robots from the same symbolic description is much easier than reusing code written for one platform by extracting its semantic meaning and regenerating the skill for another platform. Hence, it is important to find suitable levels of abstraction, and in our case we have chosen to express the guarded motions using a set of symbolic constraints. The modular system simplifies the code generation, where the user interface only exposes a subset of parameters to the user, while the JGrafchart state machine contains the calculated reference values to the controllers and coordinates the high-level execution. The external controller is responsible for the real-time sensor control which is necessary for achieving the necessary performance for assembly operations.

In future work we plan to experiment using a mobile platform running ROS together with our dual-arm robot and thus evaluate how easy it is to extend the code generation to simultaneously support other platforms. The sequence can express control structures, such as loops and if-statements, ongoing work involves adding these control structures to the task state machine as well as describing and generating the synchronization between robots.

The robustness of the generated skills depends on the user input. One direction of future work is to couple the graphical user interface with haptic demonstrations and learning algorithms in order to extract e.g., force thresholds and impedance controller parameters. Another direction is to add knowledge and reasoning to the system to automatically generate error handling states to the task state machine.

## 7 Acknowledgments

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). The third author is a member of the LCCC Linnaeus Center and the eLLIIT Excellence Center at Lund University.

The work described in this paper has been done in tight collaboration with other researchers from the project consortia. The authors are indebted for many fruitful discussions.

The authors are grateful to Anders Robertsson for careful proofreading.





# Bibliography

- [1] ABB. RobotStudio, 2015. Available from: <http://new.abb.com/products/robotics/robotstudio>. Online; last accessed 17 February 2015.
- [2] Brenna D. Argall, Sonia Chernova, Manuela Veloso, and Brett Browning. A survey of robot learning from demonstration. *Robotics and Autonomous Systems*, 57(5):469–483, 2010.
- [3] Stephen Balakirsky. Ontology based action planning and verification for agile manufacturing. *Robotics and Computer-Integrated Manufacturing*, 33, 2015.
- [4] Stephen Balakirsky, Zeid Kootbally, Craig Schlenoff, Thomas Kramer, and Satyandra Gupta. An industrial robotic knowledge representation for kit building applications. In *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, Vilamoura, Algarve, Portugal, 2012.
- [5] Michael Beetz, Lorenz Mösenlechner, and Moritz Tenorth. CRAM: A cognitive robot abstract machine for everyday manipulation in human environments. In *Proc. of IEEE/RSJ International Conference on Intelligent Robots and Systems*, Taipei, Taiwan, 2010.
- [6] George A. Bekey. *Autonomous Robots*. MIT Press, 2005.
- [7] Aude Billard, Sylvain Calinon, Rüdiger Dillmann, and Stefan Schaal. *Springer Handbook of Robotics, chapter: Robot Programming by Demonstration*, pages 1371–1394. Springer Verlag, 2008.
- [8] Aude Billard and Daniel Grollman. Robot learning by demonstration. *Scholarpedia*, 8(12):3824, 2013. revision no. 138061.
- [9] Rainer Bischoff, Arif Kazi, and Markus Seyfarth. The morpha style guide for icon-based programming. In *Proc. of the IEEE Int. Workshop on Robot and Human Interactive Communication*, 2002.
- [10] Anders Björkelund, Bernd Bohnet, Love Hafdell, and Pierre Nugues. A high-performance syntactic and semantic dependency parser. In *Coling*

## BIBLIOGRAPHY

---

- 2010: *Demonstration Volume*, pages 33–36, Beijing, China, August 23–27 2010.
- [11] Anders Björkelund, Lisett Edström, Mathias Haage, Jacek Malec, Klas Nilsson, Pierre Nugues, Sven Gestegård Robertz, Denis Störkle, Anders Blomdell, Rolf Johansson, Magnus Linderöth, Anders Nilsson, Anders Robertsson, Andreas Stolt, and Herman Bruyninckx. On the integration of skilled robot motions for productivity in manufacturing. In *Proc. IEEE International Symposium on Assembly and Manufacturing*, Tampere, Finland, 2011.
- [12] Anders Björkelund, Love Hafdell, and Pierre Nugues. Multilingual semantic role labeling. In *Proc. of the Thirteenth Conference on Computational Natural Language Learning (CoNLL): Shared Task*, pages 43–48, Boulder, Colorado, USA, 2009.
- [13] Anders Björkelund, Jacek Malec, Klas Nilsson, Pierre Nugues, and Herman Bruyninckx. Knowledge for intelligent industrial robots. In *Proc. AAAI 2012 Spring Symp. On Designing Intelligent Robots*, Stanford Univ., 2012.
- [14] Anders Blomdell, Isolde Dressler, Klas Nilsson, and Anders Robertsson. Flexible application development and high-performance motion control based on external sensing and reconfiguration of ABB industrial robot controllers. In *Proc. of ICRA 2010*, pages 62–66, Anchorage, AK, USA, 2010.
- [15] Vitor Bottazzi and Jaime Fonseca. Off-line programming industrial robots based in the information extracted from neutral files generated by the commercial CAD tools. *Industrial Robotics: Programming and Simulation and Application*, 2006.
- [16] Ronald J. Brachman and Hector J. Levesque. *Readings in Knowledge Representation*. Morgan Kaufmann, 1985.
- [17] Bride, 2014. Available from: <http://wiki.ros.org/bride>. Online; last accessed 14 January 2015.
- [18] Herman Bruyninckx and Joris De Schutter. Specification of Force-Controlled Actions in the “Task Frame Formalism”-A Synthesis. In *IEEE Transactions on Robotics and Automation*, volume 12, 1996.
- [19] Joel Luis Carbonera, Sandro Rama Fiorini, Edson Prestes, Vitor A. M. Jorge, Mara Abel, Raj Madhavan, Angela Locoro, Paulo Goncalves, Tamas Haidegger Marcos E. Barreto, and Craig Schlenoff. Defining position in a core ontology for robotics. In *Proc. 2013 IEEE/RSJ IROS*, Tokyo, Japan, 2013.
- [20] A.F. Cutting-Decelle, R.I.M. Young, J.J. Michel, R. Grangeland, J. Le Cardinal, and J.P. Bourey. ISO 15531 MANDATE: A Product-process-resource based Approach for Managing Modularity in Production Management. *Concurrent Engineering*, 15, 2007.

- [21] Joris De Schutter, Tinne De Laet, Johan Rutgeerts, Wilm Decré, Ruben Smits, Erwin Aertbeliën, Kasper Claes, and Herman Bruyninckx. Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty. 26(5):433–455, 2007.
- [22] Ayssam Elkady and Tarek Sobh. Robotics middleware: A comprehensive literature survey and attribute-based bibliography. *Journal of Robotics*, 2012, 2012.
- [23] FANUC Robotics. RoboGuide. Available from: <http://www.fanucrobotics.de/en/products/software/simulation%20and%20development/roboguide>. Online; last accessed 14 January 2015.
- [24] Charles J. Fillmore. Frame semantics and the nature of language. *Annals of the New York Academy of Sciences: Conference on the Origin and Development of Language and Speech*, 280:20–32, 1976.
- [25] Bernd Finkemeyer, Torsten Kröger, and Friedrich M. Wahl. Executing assembly tasks specified by manipulation primitive nets. *Advanced Robotics*, 19(5):591–611, 2005.
- [26] Sandro Rama Fiorinia, Joel Luis Carboneraa, Paulo Gonçalvesb, Vitor A.M. Jorgea, Vitor Fortes Reya, Tamás Haideggerd, Mara Abela, Signe A. Redfieldf, Stephen Balakirsky, Veera Ragavanh, Howard Lii, Craig Schlenoffj, and Edson Prestesa. Extensions to the core ontology for robotics and automation. *Robotics and Computer-Integrated Manufacturing*, 33:3–11, 2015.
- [27] ABB Flexible Automation. *RAPID Reference Manual*.
- [28] FrameNet. <https://framenet.icsi.berkeley.edu/fndrupal/>, 2013. Available from: <https://framenet.icsi.berkeley.edu/fndrupal/>. Online; accessed 14 January 2015.
- [29] Michael R. Genesereth and Richard E. Fikes. Knowledge interchange format, version 3.0. Technical report, Stanford University Logic Group, Technical Report Logic-92-1, 1992.
- [30] Malik Ghallab, Dana Nau, and Paolo Traverso. *Automated Planning, Theory and Practice*. Morgan-Kaufman, 2004.
- [31] Rafal Goebel, Ricardo G. Sanfelice, and Andrew R. Teel. *Hybrid Dynamical Systems: Modeling and Stability and and Robustness*. Princeton University Press, 2012.
- [32] Mathias Haage, Jacek Malec, Anders Nilsson, Klas Nilsson, and Sławomir Nowaczyk. Declarative-knowledge-based reconfiguration of automation systems using a blackboard architecture. In Anders Kofod-Petersen, Fredrik Heintz, and Helge Langseth, editors, *Proc. 11th Scandinavian Conference on Artificial Intelligence*, pages 163–172. IOS Press, 2011.

## BIBLIOGRAPHY

---

- [33] Martin Hägele, Klas Nilsson, and J. Noberto Pires. *Springer Handbook of Robotics, chapter: Industrial Robotics*, pages 963–986. Springer Verlag, 2008.
- [34] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, 1987.
- [35] Graeme Hirst. *Semantic interpretation and the resolution of ambiguity*. Cambridge University Press, 1987.
- [36] IEC. IEC 61131-3: Programmable controllers – part 3: Programming languages. Technical report, International Electrotechnical Commission, 2003.
- [37] R. Johansson and P. Nugues. Dependency-based syntactic-semantic analysis with probbank and nombank. In *Proceedings of CoNLL-2008*, pages 183–187, Manchester, United Kingdom, 2008.
- [38] Vitor A.M. Jorge, Vitor F. Rey, Renan Maffei, Sandro Rama Fiorini, Joel Luis Carbonera, Flora Branchi, João P. Meireles, Guilherme S. Franco, Flávia Farina, Tatiana S. da Silva, Mariana Kolberg, Mara Abel, and Edson Prestes. Exploring the IEEE ontology for robotics and automation for heterogeneous agent interaction. In *Robotics and Computer-Integrated Manufacturing*, volume 33, pages 12–20, 2015.
- [39] Stuart Kent. Model driven engineering. In Michael Butler, Luigia Petre, and Kaisa Sere, editors, *Integrated Formal Methods*, volume 2335 of *Lecture Notes in Computer Science*, pages 286–298. Springer Berlin Heidelberg, 2002.
- [40] Markus Klotzbücher and Herman Bruyninckx. Coordinating robotic tasks and systems with rFSM statecharts. *Journal of Software Engineering for Robotics*, 1(3):28–56, 2012.
- [41] Z. Kootbally, C. Schlenoff, C. Lawler, T. Kramer, and S.K. Gupta. Towards robust assembly with knowledge representation for the planning domain definition language (PDDL). *Robotics and Computer-Integrated Manufacturing*, 33, 2015.
- [42] I. Kresse and M. Beetz. Movement-aware action control — integrating symbolic and control-theoretic action execution. In *Proc. ICRA 2012*, pages 3245–3251, Saint Paul, MN, USA, 2012.
- [43] T. Kroeger, B. Finkemeyer, and F. M. Wahl. Manipulation primitives — a universal interface between sensor-based motion control and robot programming. In *Robotic Systems for Handling and Assembly*, pages 293–313. Springer, 2010.
- [44] KUKA. KUKA Sunrise. Available from: [http://www.kuka-labs.com/en/service\\_robotics/robot\\_control\\_system/](http://www.kuka-labs.com/en/service_robotics/robot_control_system/). Online; last accessed 14 January 2015.

- [45] KUKA System Software. *KUKA System Software 5.5 Operating and Programming Instructions for System Integrators*, 2010.
- [46] Douglas B. Lenat. Cyc: A large-scale investment in knowledge infrastructure. *Communications of the ACM*, 38(11):33–38, 1995.
- [47] Matt MacMahon, Brian Stankiewicz, and Benjamin Kuipers. Walk the talk: Connecting language, knowledge, and action in route instructions. In *Proceedings of the 21st National Conference on Artificial Intelligence - Volume 2*, AAAI'06, pages 1475–1482. AAAI Press, 2006.
- [48] J. Malec, K. Nilsson, and H. Bruyninckx. Describing assembly tasks in a declarative way. In *ICRA 2013 WS Semantics and Identification and Control of Robot-Human-Environment Interaction*, Karlsruhe, Germany, 2013.
- [49] Mitchell Marcus, Grace Kim, Mary Ann Marcinkiewicz, Robert MacIntyre, Ann Bies, Mark Ferguson, Karen Katz, and Britta Schasberger. The Penn Treebank: Annotating predicate argument structure. *ARPA Human Language Technology Workshop*, 1994.
- [50] Björn Matthias, Susanne Oberer-Treitz, and Hao Ding. Collision testing for human-robot collaboration. In *Safety in Human-Robot Coexistence and Interaction: How Can Standardization and Research benefit from each other?. IEEE Int. Conf. Intelligent Robots and Systems (IROS)*, Vilamoura, Portugal, 2012.
- [51] C. Matuszek, J. Cabral, M. Witbrock, and J. DeOliveira. An introduction to the syntax and content of Cyc. In *AAAI Spring Symposium on Formalizing and Compiling Background Knowledge and Its Applications to Knowledge Representation and Question Answering*, 2006.
- [52] Adam Meyers, Ruth Reeves, Catherine Macleod, Rachel Szekely, Veronika Zielinska, Brian Young, and Ralph Grishman. The NomBank project: An interim report. In Adam Meyers, editor, *HLT-NAACL 2004 Workshop: Frontiers in Corpus Annotation*, pages 24–31, Boston, May 2004.
- [53] S. Mitsi, K.-D. Bouzakis, G. Mansour, D. Sagris, and G. Maliaris. Off-line programming of an industrial robot for manufacturing. In *Int. J. Adv. Manuf. Technol.*, volume 26, pages 262–267, 2005.
- [54] Nader Mohamed, Jameela Al-Jaroodi, and Imad Jawhar. Middleware for robotics: A survey. In *Proc. of The IEEE Intl. Conf. on Robotics and Automation and Mechatronics (RAM 2008)*, pages 736–742, 2008.
- [55] Motoman Incorporated. *MotoSim EG Instructions*, 2007.
- [56] Neo Technology, Inc. Neo4J, 2015. Available from: <http://neo4j.com/>. Online; accessed 2 February 2015.

## BIBLIOGRAPHY

---

- [57] Pedro Neto, J. Norberto Pires, and A. Paulo Moreira. High-level programming and control for industrial robotics: using a hand-held accelerometer-based input device for gesture and posture recognition. *Industrial Robot*, 37(2):137–147, 2010.
- [58] Anders Nilsson, Riccardo Muradore, Klas Nilsson, and Paolo Fiorini. Ontology for robotics: a roadmap. In *Proc. of The Int. Conf. Advanced Robotics (ICAR09)*, Munich, Germany, 2009.
- [59] Klas Nilsson, Elin Anna Topp, Jacek Malec, and Il-Hong Suh. Enabling reuse of robot tasks and capabilities by business-related skills grounded in natural language. In *ICAS 2013, 9th Int. Conference on Autonomic and Autonomous Systems*, Lisbon, Portugal, 2013.
- [60] N. J. Nilsson. Shakey the robot. Technical Report 323, SRI International, Menlo Park, CA, USA, 1984.
- [61] Ando Noriaki, Suehiro Takashi, Kitagaki Kosei, Kotoku Tetsuo, and Yoon Woo-Keun. RT-component object model in RT-middleware — distributed component middleware for RT (robot rechnology). In *Proc. of IEEE international symposium on computational intelligence in robotics and automation (CIRA)*, pages 457–62, Espoo, Finland, 2005.
- [62] Pierre M. Nugues. *An Introduction to Language Processing with Perl and Prolog*, chapter 1, pages 1–21. Springer, 2016.
- [63] Pierre M. Nugues. *An Introduction to Language Processing with Perl and Prolog*, chapter 6, pages 148–162. Springer, 2016.
- [64] Open Source Robotics Foundation. Gazebo, 2014. Available from: <http://gazebo.sim.org/>. Online; accessed 14 January 2015.
- [65] OpenRDF Sesame, 2015. Available from: <http://rdf4j.org/>. Online; last accessed 17 February 2015.
- [66] Martha Palmer, Paul Kingsbury, and Daniel Gildea. The proposition bank: An annotated corpus of semantic roles. *Computational Linguistics*, 31(1):71–106, 2005.
- [67] Z. Pan, J. Polden, N. Larkin, S. van Duin, and J. Norrish. Recent progress on programming methods for industrial robots. In *41st International Symposium on Robotics (ISR) and 6th German Conference on Robotics (ROBOTIK)*, pages 619–626, Berlin, Germany, 2010. VDE VERLAG GMBH.
- [68] Mikkel Rath Pedersen, Dennis Herzog, and Volker Krüger. Intuitive skill-level programming of industrial handling tasks on a mobile manipulator. In *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, pages 4523–4530, Chicago, IL, USA, 2014.

- [69] Princeton University. About WordNet, 2010. Available from: <http://wordnet.princeton.edu/>. Online; last accessed 17 February 2015.
- [70] The Orocos Project. Orocos. Available from: <http://www.orocos.org/>. Online; last accessed 14 January 2015.
- [71] Hajo Rijgersberg, Mark van Assem, and Jan Top. Ontology of units of measure and related concepts. *Semantic Web Journal*, 4(1):3–13, 2013.
- [72] Rock Robotics, 2015. Available from: <http://rock-robotics.org/>. Online; last accessed 17 February 2015.
- [73] ROS, 2014. Available from: <http://wiki.ros.org/>. Online; last accessed 14 January 2015.
- [74] ROS-I-Consortium. ROS industrial. Available from: <http://rosindustrial.org/>. Online; last accessed 14 January 2015.
- [75] G. Rossano, C. Martinez, M. Hedelind, S. Murphy, and T. Fuhlbrigge. Easy robot programming concepts: An industrial perspective. In *Proceedings 9th IEEE International Conference on Automation Science and Engineering*, Madison, Wisconsin, USA, 2013.
- [76] J. Ruppenhofer, M. Ellsworth, M. R. L. Petruck, R. Johnson, and J. Schefczyk. FrameNet II: Extended theory and practice. Tech. report, UCB, 2010.
- [77] James G. Schmolze. Physics for robots. In *Proc. AAAI-86*, pages 44–50, 1986.
- [78] Nobuyuki Shimizu and Andrew R Haas. Learning to follow navigational route instructions. In *IJCAI*, volume 9, pages 1488–1493, 2009.
- [79] Smach, 2013. Available from: <http://wiki.ros.org/smach/Documentation>. Online; last accessed 14 January 2015.
- [80] Maj Stenmark and Jacek Malec. Knowledge-based industrial robotics. In *Proc. of The 12th Scandinavian AI Conference*, Aalborg, Denmark, 2013.
- [81] Maj Stenmark and Jacek Malec. Describing constraint-based assembly tasks in unstructured natural language. In *Proc. IFAC 2014 World Congress*, Cape Town, South Africa, 2014.
- [82] Maj Stenmark, Jacek Malec, Klas Nilsson, and Anders Robertsson. On distributed knowledge bases for industrial robotics needs. In *Proc. Cloud Robotics Workshop at IROS 2013*, Tokyo, Japan, 2013.
- [83] Maj Stenmark and Pierre Nugues. Natural language programming of industrial robots. In *Proc. International Symposium of Robotics 2013*, Seoul, South Korea, 2013.



## BIBLIOGRAPHY

---

- [84] Maj Stenmark and Andreas Stolt. A system for high-level task specification using complex sensor-based skill. In *RSS 2013 workshop and Programming with constraints: Combining high-level action specification and low-level motion execution*, Berlin, Germany, 2013.
- [85] A. Stolt, M. Linderoth, A. Robertsson, and R. Johansson. Adaptation of force control parameters in robotic assembly. In *10th International IFAC Symposium on Robot Control*, Dubrovnik, Croatia, 2012.
- [86] Andreas Stolt, Magnus Linderoth, Anders Robertsson, and Rolf Johansson. Force controlled assembly of emergency stop button. In *IEEE International Conference on Robotics and Automation*, Shanghai, China, 2011.
- [87] S. Tellex, T. Kollar, S. Dickerson, M. R. Walter, A. G. Banerjee, S. Teller, and N. Roy. Understanding natural language commands for robotics navigation and mobile manipulation. In *Proceedings of AAAI 2011*, 2011.
- [88] M. Tenorth, D. Nyga, and M. Beetz. Understanding and executing instructions for everyday manipulation tasks from the world wide web. In *Proc. IEEE ICRA*, Anchorage, AK, USA, 2010.
- [89] Moritz Tenorth and Michael Beetz. KnowRob – a knowledge processing infrastructure for cognition-enabled robots. In *International Journal of Robotics Research*, volume 32, 2013.
- [90] Alfred Theorin. *Adapting Grafchart for Industrial Automation*. PhD thesis, Licentiate Thesis, Lund University, Department of Automatic Control, 2013.
- [91] Alfred Theorin. *A Sequential Control Language for Industrial Automation*. PhD thesis, Lund University, 2014.
- [92] B. J. Thomas and O. C. Jenkins. RoboFrameNet: Verb-centric semantics for actions in robot middleware. In *Proc. IEEE ICRA*, Saint Paul, MN, USA, 2012.
- [93] Deutsche Gesetzliche Unfallversicherung. Bg/bgia risk assessment recommendations according to machinery directive, design of workplaces with collaborative robots. Technical report, Report no. U001/2009e, 2011.
- [94] Lund University. JGrafchart, 2013. Available from: <http://www.control.lth.se/grafchart>. Online; last accessed 17 February 2015.
- [95] Dominick Vanthienen, Markus Klotzbuecher, and Herman Bruyninckx. The 5C-based architectural composition pattern. *Journal of Software Engineering for Robotics*, 5(1):17–35, 2014.
- [96] Visual Components. 3DCreate. Available from: <http://www.visualcomponents.com/products/3dcreate/>. Online; accessed 14 January 2015.

- [97] W3C. OWL. Available from: <http://www.w3.org/TR/owl-features/>. Online; last accessed 2 February 2015.
- [98] Markus Waibel, Michael Beetz, Javier Civera, Raffaello d'Andrea, Jos Elfring, Dorian Galvez-Lopez, Kai Häussermann, Rob Janssen, J.M.M. Montiel, Alexander Perzylo, Bjoern Schiessle, Moritz Tenorth, Oliver Zweigle, and M.J.G. (René) Van de Molengraft. RoboEarth. *Robotics and Automation Magazine, IEEE*, 18(2):69–82, 2011.
- [99] T. Winograd. Procedures as a representation for data in a computer program for understanding natural language. Technical report, MIT, 1971.
- [100] Yaskawa Motoman Robotics. *DX100 OPTIONS INSTRUCTIONS FOR INFORM LANGUAGE*, 3 edition.

With more advanced manufacturing technologies, small and medium sized enterprises can compete with low-wage labor by providing customized and high quality products. For small production series, robotic systems can provide a cost-effective solution. However, for robots to be able to perform on par with human workers in manufacturing industries, they have to become flexible and autonomous in their task execution and swift and easy to instruct. This will enable small businesses with short production series or highly customized products to use robot coworkers without consulting expert robot programmers. The objective of this thesis is to explore programming solutions that can reduce the programming effort of sensor-controlled robot tasks. The robot motions are expressed using constraints, and a number of simple constrained motions can be combined into a robot *skill*. The skill can be stored in a database together with a semantic description, which enables reuse and reasoning. The main contributions of the thesis are 1) development of ontologies for robot devices and skills, 2) a user interface that provides programming support for task descriptions in unstructured natural language and 3) an implementation where low-level code is generated from the high-level descriptions. The resulting system greatly reduces the number of parameters exposed to the user. These parameters are described on a semantic level, which means that the same skill can be used on different robot platforms. The research is presented in four peer-reviewed papers. The first covers knowledge-based instruction and the system architecture. The two following papers describe the natural language programming feature of the system as well as a description of the user interface. The fourth and last paper describes the code generation step, thus connecting the high-level language instructions to real-time executable code.