LUND UNIVERSITY

From High-Level Task Descriptions to Executable Robot Code

Stenmark, Maj; Malec, Jacek; Stolt, Andreas

Published in:
Advances in Intelligent Systems and Computing

DOI:
10.1007/978-3-319-11310-4_17

2015

Link to publication

Citation for published version (APA):
Stenmark, M., Malec, J., & Stolt, A. (2015). From High-Level Task Descriptions to Executable Robot Code. In D. Filev, J. Jabłkowski, J. Kacprzyk, M. Krawczak, V. Sgurev, E. Sotirova, P. Szynkarczyk, & S. Zadrożny (Eds.), *Advances in Intelligent Systems and Computing* (Vol. 323, pp. 189-202). Springer. https://doi.org/10.1007/978-3-319-11310-4_17

Total number of authors:
3

General rights
Unless other specific re-use rights are stated the following general rights apply:
Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.
• Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
• You may not further distribute the material or use it for any profit-making activity or commercial gain
• You may freely distribute the URL identifying the publication in the public portal

Read more about Creative commons licenses: https://creativecommons.org/licenses/

Take down policy
If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

LUND UNIVERSITY

PO Box 117
221 00 Lund
+46 46-222 00 00

Download date: 05. Dec. 2025

From High-Level Task Descriptions to Executable Robot Code

Maj Stenmark[1], Jacek Malec[1], and Andreas Stolt[2]

[1] Dept. of Computer Science, Lund University, 221 00 Lund, Sweden
`maj.stenmark@cs.lth.se`, `jacek.malec@cs.lth.se`
[2] Dept. of Automatic Control, Lund University,221 00 Lund, Sweden
`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 [1–4].

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 [1, 9] may be considered in light of the Model-Driven Engineering principles [5]. 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 [6]. It consists of the following components:

- An intuitive task-definition tool that allows the user to specify the task using graphical menys and downloading assembly skills from a knowledge base, or by using a natural-language interface [4, 7];
- 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` [8] for the ABB industrial robots, and ROS-based (`www.ros.org`) for the Rob@Work mobile platform;
- ABB robots: a dual-arm concept robot, IRB120 and IRB140, Rob@Work platform from Fraunhofer IPA (`http://www.care-o-bot.de/en/rob-work.html`), Force/Torque sensors from ATI Industrial Automation (`http://www.ati-ia.com`) 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 [2, 3, 9]. 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 (ABB

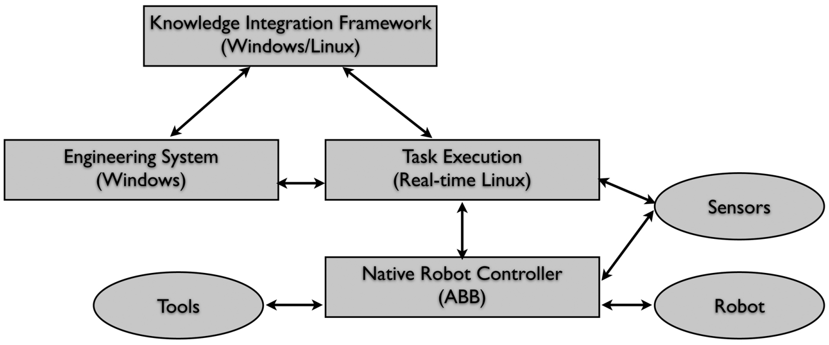
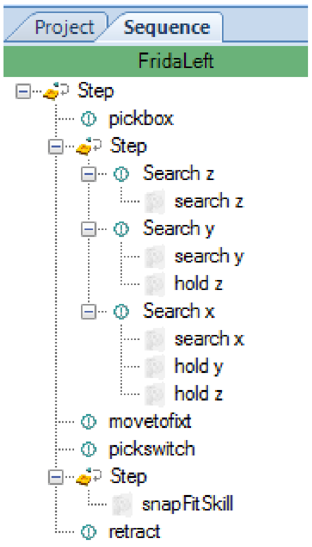

Knowledge Integration Framework (Windows/Linux)
Engineering System (Windows)
Task Execution (Real-time Linux)
Sensors
Native Robot Controller (ABB)
Tools
Robot

Fig. 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.

RobotStudio [11]) 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 [12], used both as a sequential function chart [13]—a variant of Statecharts [14]—editor, and its execution environment), can be parsed, automatically or manually annotated with semantic data, and stored in the skill libraries.

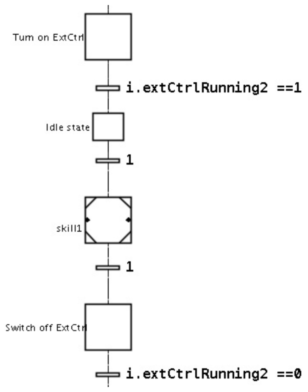
The Task Execution module is built on top of the native robot controller and sensor hardware. It compiles, with the help of KIF, a symbolic task specification (like the one shown in Fig. 2) into generic executable files and, when needed, hardware-specific code, before executing it. It is implemented on a real-time-enabled Linux machine, linking the external control coming from JGrafchart (a simple example is shown in Fig. 2(b)) 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 [15] translating a symbolic, constraint-based [16] 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

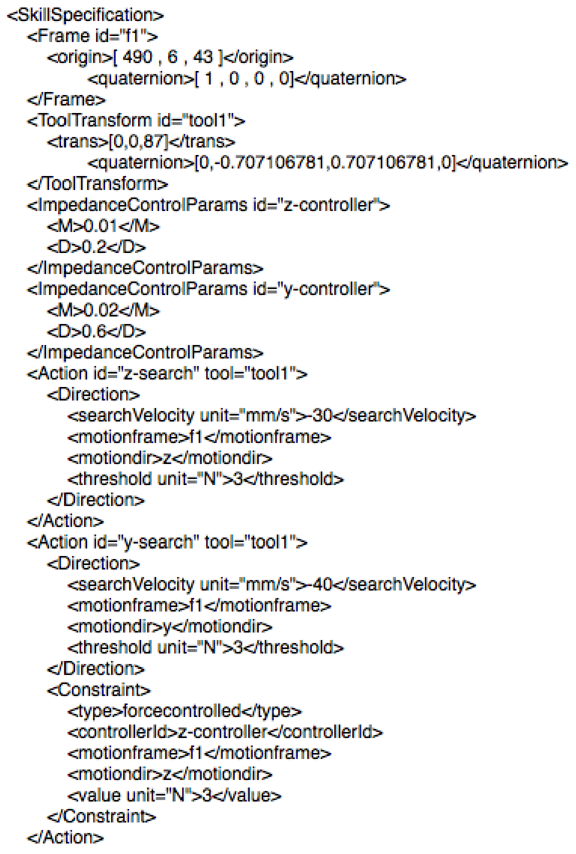


```
FridaLeft
Step
    pickbox
    Step
        Search z
            search z
        Search y
            search y
            hold z
        Search x
            search x
            hold y
            hold z
    movetofixt
    pickswitch
    Step
        snapFitSkill
    retract
```

(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. 2(a).



```
<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>
```

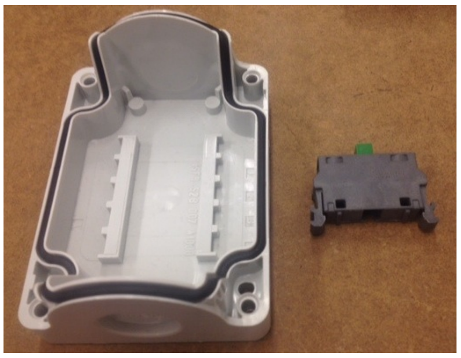
(c) A sample XML description corresponding to the guarded motion skill from Fig. 2(a) that is sent to the code generation service by Engineering System. The parameter values are either set automatically or by the user in the Engineering System. If a guarded motion is generated, e.g., from text and one 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.

Fig. 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. 2(a). In Fig. 2(b) 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. 2(c) presents the corresponding input to the code generation service.

services can be treated as black boxes. Robot and system-integration vendors can offer their customers computationally expensive or data-heavy cloud-based services [10] 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. 3(a). The task is described in the Engineering System as a sequence, shown earlier in Fig. 2(a). 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 code for tasks containing these three types of actions. As an example we will use the sequence shown in Fig. 2(a) that, when executed, requires switching between the native robot controller and the external, sensor-based control (`ExtCtrl`).


(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.

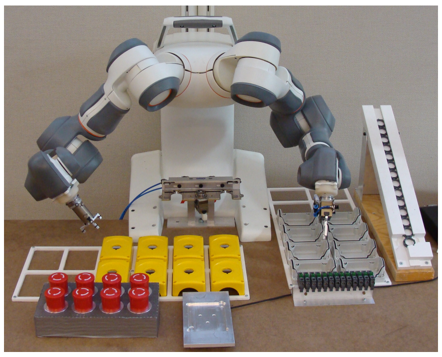

(b) The two-armed ABB robot and the workspace setup.

Fig. 3. The example setup for the assembly experiments.

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.

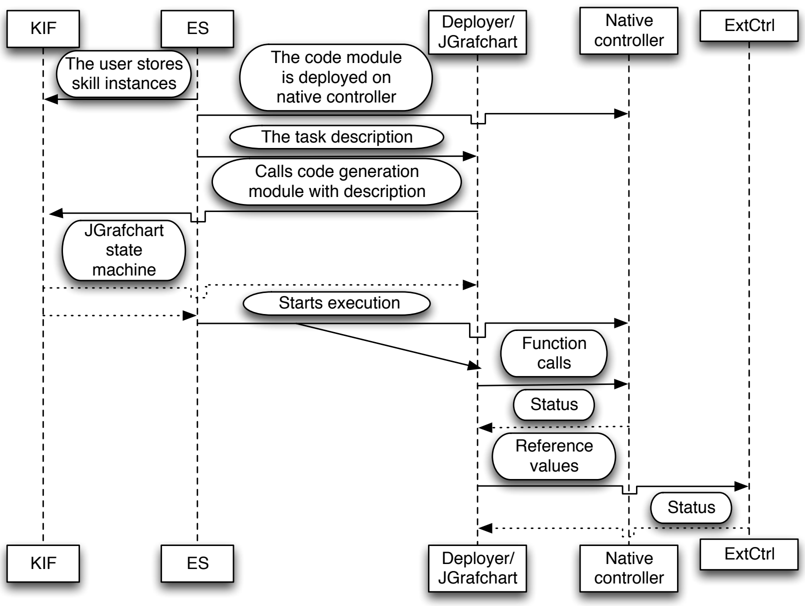


Fig. 4. The Engineering System (ES) sends the task description to a small helper program called Deployer which in turn calls the code generation service on KIF, loads the returned file and starts JGrafchart.

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 [16], 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

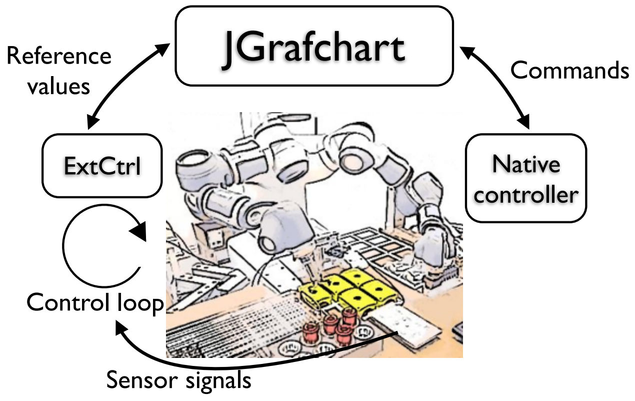


Fig. 5. A schematic image of the execution architecture. The task state machine is executed in JGrafchart, which in turn sets and reads reference values to `ExtCtrl` and sends commands to the native controller.

generated for multiple languages (C, Java, RAPID, C#). The `ExtCtrl` interface divides the 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 [12]. 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. 2(b) 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. 2(c). 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.

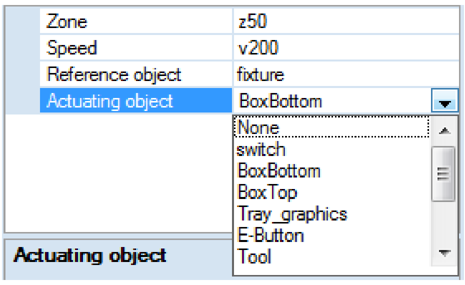
3.5 Guarded motions

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. 2(c), 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.

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



Zone	z50
Speed	v200
Reference object	fixture
Actuating object	BoxBottom

Fig. 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.

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. 2(a) which resulted in an executable state machine, the same that is partly shown in Fig. 2(b). 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. 3(b)) 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 [17, 18]. 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 [19].

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 [13] 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 [20] (`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 [21–23]. 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 [24] 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 [25]. 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 [26] 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 [27]. 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 [28]. 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 [29].

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.

References

1. Anders Björkelund, Lisett Edström, Mathias Haage, Jacek Malec, Klas Nilsson, Pierre Nugues, Sven Gestegård Robertz, Denis Störkle, Anders Blomdell, Rolf Johansson, Magnus Linderoth, 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. doi: 10.1109/ISAM.2011.5942366.
2. Jacek Malec, Klas Nilsson, and Herman Bruyninckx. *Describing assembly tasks in a declarative way.* In: ICRA 2013 WS on Semantics, Identification and Control of Robot-Human-Environment Interaction, 2013.
3. Maj Stenmark and Jacek Malec, *Knowledge-Based Industrial Robotics*, In *Proc. of The 12th Scandinavian AI Conference*, Aalborg, Denmark, November 20–22, 2013. doi: http://dx.doi.org/10.3233/978-1-61499-330-8-265

4. Maj Stenmark and Jacek Malec, Describing constraint-based assembly tasks in unstructured natural language In: Proc. IFAC 2014 World Congress, Capetown, South Africa, 24-29 August 2014.
5. Stuart Kent, *Model Driven Engineering*, In: LNCS 2335, pp. 286–298, 2002
6. Dominick Vanthienen, Markus Klotzbuecher, and Herman Bruyninckx, *The 5C-based architectural Composition Pattern*, JOSER, vol. 5, no. 1, pp. 17–35, 2014
7. Maj Stenmark and Pierre Nugues, *Natural Language Programming of Industrial Robots*, In Proc. International Symposium of Robotics 2013, Seoul, South Korea, October 2013.
8. 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, pp. 62-66, Anchorage, USA, 2010.
9. Anders Björkelund, Jacek Malec, Klas Nilsson, Pierre Nugues and Herman Bruyninckx. *Knowledge for Intelligent Industrial Robots,* Proc. AAAI 2012 Spring Symp. On Designing Intelligent Robots, Stanford Univ., March 2012.
10. 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, 3rd November 2013, `http://www.roboearth.org/wp-content/uploads/2013/03/final-13.pdf`
11. ABB RobotStudio, http://new.abb.com/products/robotics/robotstudio. Visited 2013-02-04.
12. Alfred Theorin, Adapting Grafchart for Industrial Automation, Licentiate Thesis, Lund University, Department of Automatic Control, 2013
13. IEC. IEC 61131-3: Programmable controllers – part 3: Programming languages. Technical report, International Electrotechnical Commission, 2003.
14. David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8:231–274, 1987.
15. Maj Stenmark and Andreas Stolt. A System for High-Level Task Specification Using Complex Sensor-based Skill In *RSS 2013 workshop, Programming with constraints: Combining high-level action specification and low-level motion execution*, Berlin, Germany, 2013.
16. 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. *The International Journal of Robotics Research*, 26(5):433–455, 2007.
17. 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. VDE VERLAG GMBH, Berlin, 2010.
18. 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.
19. A. Billard, S. Calinon, R. Dillmann, and S. Schaal. *Springer Handbook of Robotics*, chapter: Robot Programming by Demonstration, pages 1371–1394. Springer Verlag, 2008.
20. Andres Nilsson, Riccardo Muradore, Klas Nilsson and Paolo Fiorini, *Ontology for Robotics: a Roadmap*, Proceedings of The Int. Conf.Advanced Robotics (ICAR09), Munich, Germany, 2009.

21. S. Mitsi, K.-D.Bouzakis, G. Mansour, D. Sagris, and G. Maliaris. Off-line programming of an industrial robot for manufacturing Int. J. Adv. Manuf. Technol., (2005) 26:262-267
22. 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, Simulation and Application, Edited by: Low Kin Huat, ISBN 3-86611-286-6, 2006.
23. M. Hägele, K. Nilsson, and J. N. Pires. *Springer Handbook of Robotics*, chapter: Industrial Robotics, pages 963–986. Springer Verlag, 2008.
24. R. Bischoff, A. Kazi, and M. Seyfarth. The morpha style guide for icon-based programming. In *Proc. of the IEEE Int. Workshop on Robot and Human Interactive Communication*, 2002.
25. 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, pp. 137–147, Vol. 37, No. 2, 2010.
26. Andreas Stolt, Magnus Linderoth, Anders Robertsson, and Rolf Johansson. Force controlled assembly of emergency stop button. In *2011 IEEE International Conference on Robotics and Automation*, Shanghai, China, May 2011.
27. 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.
28. 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, October 18–22, 2010, Taipei, Taiwan.
29. I. Kresse and M. Beetz. Movement-Aware Action Control Integrating Symbolic and Control-Theoretic Action Execution. In Proc. ICRA 2012, pages 3245–3251, 2012.