Composition and Incremental Refinement of Skill Models For Robotic Assembly Tasks

Frank Nägele, Lorenz Halt, Philipp Tenbrock
Fraunhofer Institute for Manufacturing Engineering and Automation IPA
Stuttgart, Germany
e-mail: {frn, lth, pgt}@ipa.fraunhofer.de

Andreas Pott
Institute for Control Engineering of Machine Tools and Manufacturing Units
University of Stuttgart, Germany
e-mail: andreas.pott@isw.uni-stuttgart.de

***Abstract*—Skill-based approaches for programming robots promise many benefits such as easier reuse of functionality across applications, encapsulation and hiding of process details, and often a certain level of hardware abstraction. While many previous approaches allow to compose high-level skills, the basic skills themselves are usually atomic, non-extensible, or are supposed to be created by experts beforehand.**

We designed a prototype-based skill model to create force-controlled manipulation skills for robotic assembly tasks. In this paper, we show how skills can be created by composing sub-elements, each modeling a different aspect of a skill such as kinematics, task description, or coordination. Using prototype-based inheritance, skills can be incrementally refined and extended to create new skills, instead of creating each skill from scratch.

Our skill model uses iTaSC for task specification, hierarchical statecharts for coordinating the execution of skills, and provides a domain-specific language that allows to quickly compose and parameterize skills and applications.

I. Introduction

Skill-based approaches for programming robots have been a subject of research for almost three decades, as skills promise many benefits over the traditional robot programming approach of teaching and replaying trajectories. Complex processes are broken down into smaller and easier to handle sub-steps, each being executed by a single skill. Skills can provide parameters, so users are able to configure the skills according to the task at hand. When skills are designed with a certain generality, they can be reused across a wide range of applications, saving programming efforts.

Hasegawa et al. [1] describe one of the first systems using model-based manipulation skills. By explicitly modeling the environment, a wide range of changes can be represented by parameters, making the system more adaptable than conventional approaches. A hybrid position/velocity/force control scheme helps to cope with uncertainties in the models. The newly introduced complexities of sensor-based robot control are fully encapsulated in the manipulation skills and hidden from the user.

A variety of models for manipulation skills are structured similarly. Many of these models are based on the *Task Frame Formalism* (TFF) [2]. The TFF defines a *task frame* (or compliance frame) and allows to assign different control modes to each of its axes. Other models use the *Task Function Approach* (TFA) [3] or iTaSC (*instantaneous Task Specification using Constraints*) [4], which are not limited to a single task frame like the TFF. The execution of the individual skills is often coordinated by statecharts or Petri nets. Most models provide *domain-specific languages* (DSLs) as support for creating applications. The following paragraphs introduce skill models that are similar to ours.

Bruyninckx and De Schutter [5] give a formal definition for task descriptions based on the TFF and show many examples that are quite similar to some of our basic skills, including sliding, inserting, and the guarded approach. Klotzbücher et al. [6] provide DSLs for modeling skills using the TFF and for modeling a *restricted Finite State Machine* (rFSM) to coordinate their execution. Weidauer et al. [7] use place/transition (Petri) nets to coordinate and hierarchically structure *Manipulation Primitives* [8] that are also based on the TFF. Thomas et al. [9] port the Manipulation Primitives to a KUKA LWR and provide a DSL called *LightRocks* using UML/P statecharts to create skills.

Kresse et al. [10] map symbolic high-level task descriptions (e.g. 'point-towards' or 'keep-horizontal') to task functions and solve them using the TFA. Individual tasks can be combined to form complex skills. Vanthienen et al. [11] present a DSL to model skills using iTaSC. For coordination, rFSMs are used. Here as well, the composition of tasks is shown.

There are, however, certain shortcomings of previous approaches. Basic skills are often seen as atomic, non-extensible, or are supposed to be created by experts beforehand, while the actual users only arrange and parameterize them. There is often little tooling and modeling support for creating these low-level skills, therefore requiring a lot of robotics and programming expertise. Furthermore, while these skills can mostly be combined into high-level skills, it is usually only possible to run skills successively, not concurrently, in particular when skills have conflicting tasks. These issues limit the reuse of skills and their effectiveness.

In our previous work [12], we describe a prototype-based skill model that allows to create, compose, and refine force-controlled manipulation skills with a strong focus on reuse. Our model is based on iTaSC, a task specification formalism permitting the composition of partial task descriptions, as is necessary for our purpose. This paper further elaborates on the aspects of composition and incremental refinement of skills

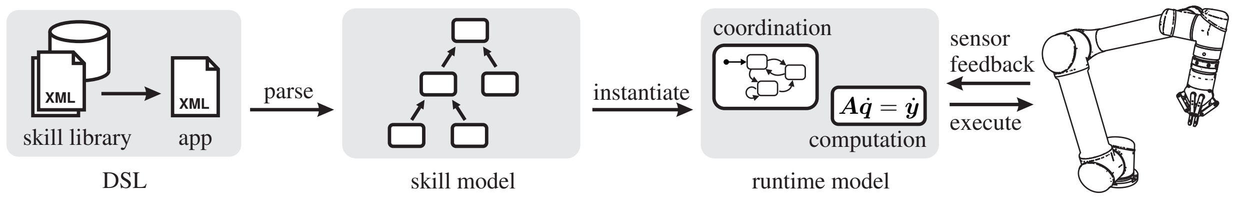


Fig. 1: Schematic overview of our system

and describes our domain-specific language in more detail. We demonstrate the high reusability of our skills by the creation of a comprehensive skill library, where new skills do not have to be built from scratch but can reuse and extend existing skills.

II. Skills for Robotic Assembly Tasks

Fig. 1 outlines the main parts of our system [12]. Skills and applications are described in a DSL with XML as a carrier syntax. Before execution, they are parsed and expressed in our prototype-based skill model. The statecharts coordinating the skills, the algorithms to execute them, and all other elements in the model are then instantiated and executed. The desired robot motions are calculated at a constant rate between 100 and 250 Hz, depending on the robot, and sent to the robot via a joint velocity interface.

Skills take on multiple roles: They contain the *task description*, modeling a sub-process to be executed by the robot. They are responsible for *coordination*, as they act as states in a hierarchical statechart and contain stop conditions (monitors) for their tasks. Lastly, they act as containers for the *kinematic elements* that are used for the task description.

We use iTaSC [4] as a task specification formalism, which allows us to compose partial task descriptions. For iTaSC, *virtual kinematic chains* (VKCs) model relative motions between robot tools and workpieces. VKCs define *feature coordinates*, which are expressed by Cartesian, cylinder, or spherical coordinates, depending on the application at hand, to make the task description as simple as possible. These feature coordinates are put into relation to the robot's joint coordinates by forming *kinematic loops* that connect the VKCs with the chain of robot links. Tasks are then described by imposing constraints on the feature and/or joint coordinates, i.e. specifying desired values for them and assigning controllers. A task description may constrain only some of the degrees of freedom and multiple partial task descriptions can be combined and executed concurrently to form more complex tasks. Each partial task formulates a sub-goal like applying a force in a certain direction or keeping a certain distance to an object. Put together, complex tasks such as a sliding motion or a peg-in-hole insertion are described. As constraints may express conflicting tasks, they are strictly prioritized and the *task priority strategy* is applied recursively as described by [13].

III. A Prototype-Based Parameter Model

In our skill model, everything is represented by a parameter, from a basic integer value to a complex skill containing many sub-parameters. In this section, we introduce the different types of parameters, describe how parameters can inherit properties from other parameters using a prototype-based inheritance scheme, and show the transformation of parameters into executable code.

As an introductory example, Listing 1 shows the definition of a parameter model for the `control_error_monitor`. It contains some meta data, defining how it should be instantiated for execution, and in its data field a list of parameters that will be passed as arguments for the initialization.

Our model applies a prototype-based inheritance scheme, which is related to JavaScript's prototype chain. In contrast to the more common class-based programming, there are no classes and new objects are always created by cloning existing ones. In our DSL, the `clone` element creates a new parameter, setting the name with the `id` attribute and the prototype to be cloned with the `prototype` attribute. For example, a simple `string` parameter can be defined by cloning the `string` prototype, simultaneously setting its new value:

```
<clone id="event" prototype="string">succeeded</clone>
```

The `type` element provides a second way to create a parameter, allowing to edit a parameter's meta data, as used in Listing 1 for the `event` parameter. Both styles clone elements the same way. The only difference is whether the meta data should be edited or not. Listing 1 also shows how existing parameters are edited using the `member` element, in this case setting the `description` string of the parameter, which is part of the meta data.

A. Types of parameters

A *basic parameter* contains a data field for one of the basic data types *string*, *float*, *int*, or *bool*. Examples are the `samples`, `event`, and `description` parameters that are shown in Listing 1. For the basic types, it is also possible to provide a list of comma-separated values, as shown for the `coordinates` and `errors` parameters. Default values may be given for the parameters or they can be left undefined. A *dictionary* is an associative container. It contains other parameters, which can be accessed using their `id`. New parameters can be inserted but each `id` has to be unique within the parameter. The `control_error_monitor` parameter itself is a dictionary. A *list* contains an ordered number of sub-parameters. The `sequence_skill`, as described later, contains a list of sub-skills. A *nested parameter* contains a single other parameter of a specified type. An example is shown later in Listing 3 where a controller is added to a `controller_assignment` using

Listing 1: Definition of a monitor and its meta data

```
<type id="control_error_monitor" prototype="monitor">

 <!-- Meta data for instantiation -->
 <meta>
   <member id="description">Checks control errors</member>
   <member id="implementation">
     <clone prototype="python">
       <member id="module">
         component_library.monitors.basic</member>
       <member id="class">ControlErrorMonitor</member>
     </clone>
     <clone prototype="orocos">
       <member id="package">monitors</member>
       <member id="component">ControlErrorMonitor</member>
     </clone>
   </member>
 </meta>

 <!-- Arguments for initialization -->
 <data>
   <clone id="coordinates" prototype="string_csv"/>
   <clone id="errors" prototype="float_csv"/>
   <clone id="samples" prototype="int">10</clone>
   <type id="event" prototype="string">
    <meta>
      <member id="description">Name of the event</member>
    </meta>
    <data>succeeded</data>
   </type>
 </data>

</type>
```

the `controller` parameter. All our model's parameters are ultimately derived from or composed of these four primitive types.

B. *Inheriting parameters*

In addition to directly cloning one of the primitive types, it is possible to use any other existing parameter as a prototype. In the example of Listing 1, the `control_error_monitor` parameter inherits from `monitor`. Cloned parameters inherit all sub-parameters from their prototypes. Editing a parameter sets a new value for the parameter, shadowing its prototype's original value. This works in the same way for basic and nested parameters, as well as for a sub-parameter in a dictionary or list.

C. *From parameter models to execution*

For instantiating and executing a model, the meta data include information on how to instantiate a parameter, as shown in Listing 1. For Python, the name of a module and the name of a class are given. The module is loaded and an object of the class is instantiated. Similarly, for our C++ implementation, a package name and the name of an OROCOS [14] component to be deployed are given. The parameters in the data field are converted to the respective programming language's native types and passed as arguments to initialize the newly created object.

IV. Composition and Incremental Refinement of Skill Models

While our parameter model allows to create, refine, and compose any type of parameter, the primary use is for defining skill models. This section describes how our extensive library

Listing 2: A skill with Cartesian feature coordinates

```
<clone id="cartesian" prototype="skill">
 <clone id="tool" prototype="frame"/>
 <clone id="target" prototype="frame"/>

 <member id="kinematic_elements">
  <clone prototype="kinematic_loop">
    <member id="robot_chains">
      <reference reference_id="robot.chain"/>
      <clone id="ch_tool" prototype="cartesian_chain">
        <member id="coordinate_type">object</member>
        <member id="base" reference_id="robot.ee_link"/>
        <member id="tip" reference_id="tool"/>
      </clone>
    </member>
    <member id="feature_chains">
      <clone id="ch_target" prototype="cartesian_chain">
        <member id="coordinate_type">object</member>
        <member id="base" reference_id="robot.base_link"/>
        <member id="tip" reference_id="target"/>
      </clone>
      <clone id="ch_feature" prototype="cartesian_chain">
        <member id="coordinate_type">feature</member>
        <member id="base" reference_id="target"/>
        <member id="tip" reference_id="tool"/>
      </clone>
    </member>
  </clone>
 </member>
</clone>
```

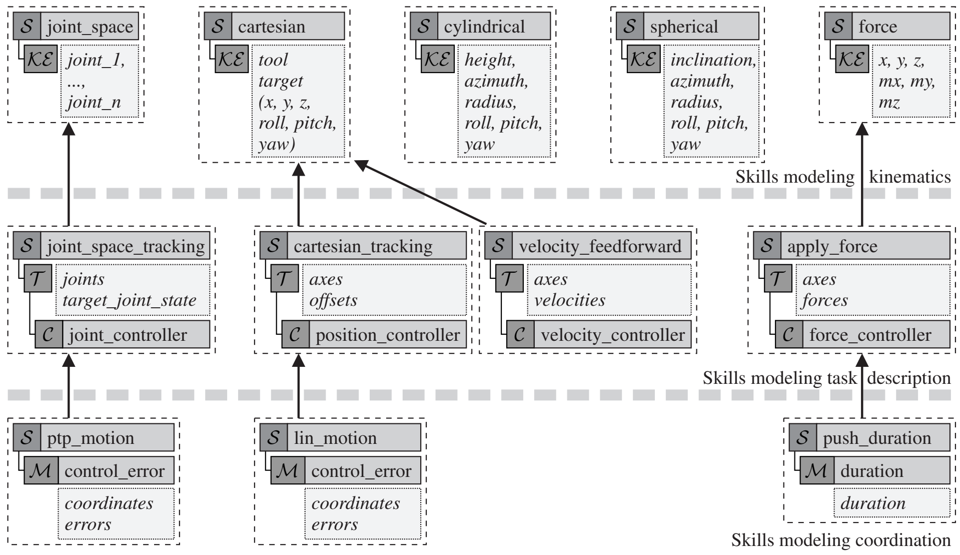
of skills is modeled. Not a single line of 'glue code' has to be written when composing these skills, everything is modeled and composed using our DSL.

A skill is essentially a container for other elements. The list of kinematic elements $\mathcal{KE}$ contains the skill's kinematic loops, chains, and links, which define its controllable coordinates following the iTaSC formalism. Tasks $\mathcal{T}$ impose constraints on these coordinates and assign controllers $\mathcal{C}$. The monitors $\mathcal{M}$ define stop conditions for a skill. They raise events that can be used for defining transitions. Each transition links one event of the skill to the skill to be executed next. Scripts $\mathcal{SC}$ fulfill supporting functions as will be described below. Lastly, the list of sub-skills allows to create hierarchically nested statecharts. Sub-skills can run in sequence, concurrently or can be arranged in a statechart.

A. *Composing and refining basic skills*

In this section, several skills of our library are built from the ground up, showing how to create new skills by incrementally adding functionality to existing skills. Exemplarily, a skill that models the kinematics for Cartesian feature coordinates is extended in multiple steps by adding tasks, monitors and scripts.

Kinematics: Listing 2 shows the definition of the `cartesian` skill. A kinematic loop is closed from the robot's base link via its end-effector to the tool frame and from the base link via the target frame to the tool frame [12]. The `tool` and `target` frames are given as parameters to the skill. These parameters are referenced by the `cartesian_chain` elements defining the loop. They provide the feature coordinates $x, y, z, roll, pitch, yaw$. While this skill does not yet contain a task description or monitors, it is the basis for all skills that use Cartesian coordinates.



$\mathcal{S}$ joint_space
$\mathcal{KE}$ joint_1, ..., joint_n
$\mathcal{S}$ cartesian
$\mathcal{KE}$ tool target (x, y, z, roll, pitch, yaw)
$\mathcal{S}$ cylindrical
$\mathcal{KE}$ height, azimuth, radius, roll, pitch, yaw
$\mathcal{S}$ spherical
$\mathcal{KE}$ inclination, azimuth, radius, roll, pitch, yaw
$\mathcal{S}$ force
$\mathcal{KE}$ x, y, z, mx, my, mz
Skills modeling kinematics
$\mathcal{S}$ joint_space_tracking
$\mathcal{T}$ joints target_joint_state
$\mathcal{C}$ joint_controller
$\mathcal{S}$ cartesian_tracking
$\mathcal{T}$ axes offsets
$\mathcal{C}$ position_controller
$\mathcal{S}$ velocity_feedforward
$\mathcal{T}$ axes velocities
$\mathcal{C}$ velocity_controller
$\mathcal{S}$ apply_force
$\mathcal{T}$ axes forces
$\mathcal{C}$ force_controller
Skills modeling task description
$\mathcal{S}$ ptp_motion
$\mathcal{M}$ control_error
coordinates errors
$\mathcal{S}$ lin_motion
$\mathcal{M}$ control_error
coordinates errors
$\mathcal{S}$ push_duration
$\mathcal{M}$ duration
duration
Skills modeling coordination

Fig. 2: Skills modeling kinematics, task description, and coordination

Listing 3: Extension of a skill to add a task description

```
<clone id="cartesian_tracking" prototype="cartesian">
 <clone id="axes" prototype="string_csv"/>
 <clone id="offsets" prototype="float_csv"/>

 <member id="tasks">
  <clone id="tracking_task" prototype="task">
    <member id="coordinates" reference_id="axes"/>
    <member id="desired" reference_id="offsets"/>
    <member id="controller_assignments">
      <clone prototype="controller_assignment">
        <member id="coordinates">x, y, z</member>
        <member id="controller">
          <clone prototype="position_controller">
            <!-- ... controller parameters -->
          </clone>
        </member>
      </clone>
      <!-- ... controller for orientation -->
    </member>
  </clone>
 </member>
</clone>
```

Task Description: As shown in Listing 3, the `cartesian_tracking` skill extends the `cartesian` skill and adds a task description to (partly) align the `tool` and `target` frames. The `axes` to be aligned are given, with an optional `offset` for each axis. Controllers are provided for controlling position and orientation and are assigned to the respective axes.

Monitors: Lastly, a monitor is added, which acts as a stop condition for the skill. Listing 4 shows the `lin_motion` skill, which extends the `cartesian_tracking` skill and adds the `control_error_monitor` introduced earlier. The monitor terminates the skill as soon as the control error is below the specified threshold for a certain number of control cycles. Listing 4 also shows how the skill is used. The parameters are exemplarily set to position the gripper frame 10 cm above a component frame, while leaving the other axes unconstrained.

Similar families of skills are created for other types of feature coordinates, e.g. robot joint coordinates, or cylindrical and spherical coordinates as indicated in Fig. 2. Other controllers can be used for velocity and force controlled skills, as in the case of the `velocity_feedforward` and `apply_force` skills.

Listing 4: Extension of a skill to add a stop condition

```
<clone id="lin_motion" prototype="cartesian_tracking">
 <member id="monitors">
   <clone id="monitor" prototype="control_error_monitor">
     <member id="coordinates" reference_id="axes"/>
     <member id="errors">
       0.001, 0.001, 0.001, 0.005, 0.005, 0.005
     </member>
   </clone>
 </member>
</clone>

<!-- Using the lin_motion skill -->
<clone id="my_lin_motion" prototype="lin_motion">
  <member id="tool">grip_point</member>
  <member id="target">component_1</member>
  <member id="axes">z</member>
  <member id="offsets">0.1</member>
</clone>
```

Listing 5: Extension of a skill to hold the current tool pose

```
<clone id="hold_pose" prototype="cartesian_tracking">
  <member id="target">hold_temp_frame</member>
  <member id="scripts">
    <clone prototype="temp_frame_script">
      <member id="frame" reference_id="target"/>
      <member id="source" reference_id="tool"/>
      <member id="parent" reference_id="robot.base_link"/>
    </clone>
  </member>
</clone>
```

Scripts: Additional functionality can be offered by scripts. The `hold_pose` skill in Listing 5 and Fig. 3 shows the typical usage of a script. The `temp_frame_script` copies the `tool` frame when starting the skill, naming it `hold_temp_frame` and setting its parent frame to the robot's base link to make it static. The `target` frame inherited from the `cartesian_tracking` skill is set to this copied frame. The robot accordingly keeps the tool at the position it had when starting the skill. When the skill is stopped, the script

Listing 6: Definition and usage of the approach skill

```
<clone id="approach" prototype="sequence_skill">
  <clone id="tool" prototype="frame"/>
  <clone id="target" prototype="frame"/>
  <clone id="offsets" prototype="float_csv"/>

  <member id="skills">
    <clone prototype="lin_motion">
      <member id="tool" reference_id="tool"/>
      <member id="target" reference_id="target"/>
      <member id="offsets" reference_id="offsets"/>
    </clone>
    <clone prototype="lin_motion">
      <member id="tool" reference_id="tool"/>
      <member id="target" reference_id="target"/>
    </clone>
  </member>
</clone>

<!-- Simple usage of the approach skill -->
<clone id="my_approach" prototype="approach">
  <member id="tool">grip_point</member>
  <member id="target">component_1</member>
  <member id="offsets">0, 0, 0.1, 0, 0, 0</member>
</clone>
```

deletes the temporary frame again. The `relative_lin` skill works similarly. It extends the `lin_motion` skill and also adds a `temp_frame_script` script. Using the `offsets` parameter inherited from the `lin_motion` skill, the tool is positioned relative to the position it had when the skill was started. Details aside, these examples show the flexibility of being able to compose skills of small, modular building blocks. They also show the reusability of these building blocks.

Some skills, such as the `idle` skill, do not specify a task. They are often combined with scripts or monitors as shown in Fig. 4: The `idle_duration` skill adds a monitor that measures the skill's execution time. After a specified duration, the monitor terminates the skill. The `set_output` skill adds a script to write to a robot's digital output. The `wait_for_input` skill idles until a robot's digital input is set, e.g. to wait for signals from peripheral equipment like an automatic screwdriver. We implemented further skills this way that tare sensors, control robot grippers, or communicate with third-party software such as vision systems.

B. Composing sequence skills

More complex skills can be created by composing skills. Using a `sequence_skill` is the easiest way to do this, as its sub-skills simply run in a consecutive order. The `approach` skill shown in Listing 6 combines two `lin_motion` skills to align the `tool` and `target` frames in two steps. This may be beneficial for approaching a part to be picked from above to prevent collisions. For the first `lin_motion` skill, an `offset` is given. As configured in this example, the tool frame is first positioned 10 cm above the target, then moves down to the actual target position.

Combining skills this way helps to encapsulate and hide process details and reduces the number of parameters to be specified by the user. Having to specify less parameters results in less effort for programming, less effort for later changes, and a smaller chance for making mistakes.

C. Composing concurrency skills

The ability of iTaSC to handle partial task descriptions allows our system to run multiple skills simultaneously. A `concurrency_skill` executes all tasks of its sub-skills at the same time and prioritizes them according to their order in the list of sub-skills. By combining position, velocity, and force-controlled skills, as well as monitors and scripts, a skill with a certain desired behavior can be created in a straightforward way.

A first example is the `guarded_lin` skill shown in Fig. 5. It contains a `lin_motion` skill and adds a `force` skill. While the latter does not contain an additional task, it provides force measurements transformed to a point of interest, which allows to add a `threshold_monitor` that terminates the skill's execution when the forces exceed a predefined limit. This skill is useful when a specific reaction to undesired forces is required, e.g. when moving to a pick position where the part to be gripped may not always be placed correctly.

The `guarded_approach` skill combines velocity and position control. It contains a `velocity_feedforward` skill, which moves the robot in a certain direction and a `hold_pose` skill with a lower priority, which controls all remaining degrees of freedom to keep the initial position. Similar to the `guarded_lin` skill, forces are measured and the skill terminates when they exceed a predefined limit. This skill is usually applied in assembly scenarios to establish the first contact between two parts. It is often followed by a `slide` skill, which combines velocity, force and position control. The `velocity_feedforward` skill is used to move the tool in one direction, while `apply_force` is keeping the contact that has previously been established, as shown in Fig. 6 for a top-hat rail assembly of electronic equipment. The `hold_pose` skill, with the lowest priority, again controls all remaining degrees of freedom to keep their initial positions. The `guarded_slide` skill adds a stop condition, which is monitoring forces in the direction of the sliding motion, to stop the robot when the contact in the second direction is established.

D. Composing statechart skills

Statechart skills allow to model complex behavior. As an example, Fig. 7 shows the logic of a screwdriving application in the form of a statechart. Assuming the screwdriver is correctly placed in a fixture, the initial state (skill) picks up the screwdriver. The statechart reflects that a screwdriver cannot be picked up again before placing it back into the fixture first. However, multiple screwing processes can be run before placing the screwdriver back. The screwdriving skills are created with the same mechanism for composing skills that is presented here. More details on them as well as other applications of the skill model are given in [12].

V. Discussion and Conclusions

In this paper, we presented our prototype-based skill model and gave an introduction to composing skills with our domain-specific language. Several aspects of our model help to achieve a high degree of reusability when creating new skills and

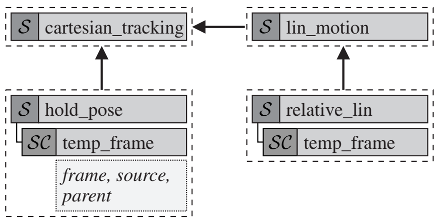


Fig. 3: Adding scripts to skills

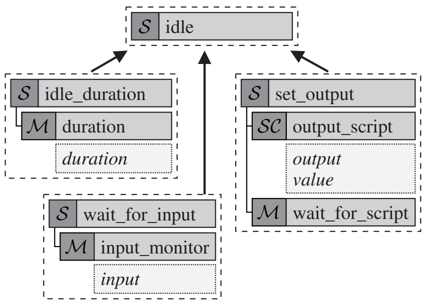


Fig. 4: Idle skills

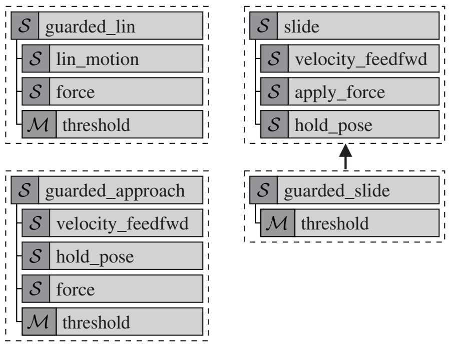
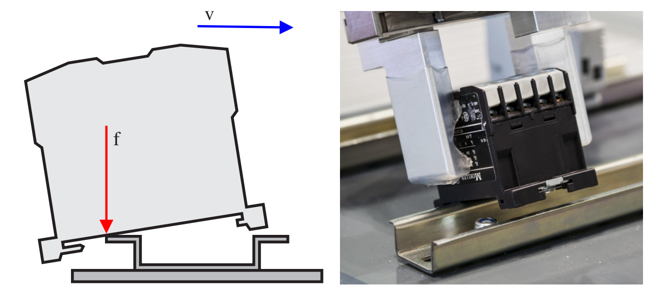
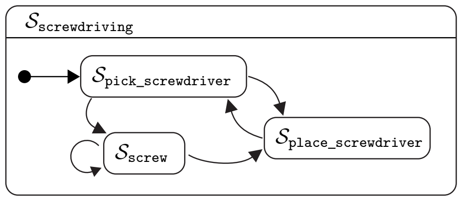


Fig. 5: Concurrency skills



(a) Directions of force and velocity (b) Top-hat rail assembly

Fig. 6: Slide skill used for assembly



$\mathcal{S}_{\texttt{screwdriving}}$
$\mathcal{S}_{\texttt{pick\_screwdriver}}$
$\mathcal{S}_{\texttt{place\_screwdriver}}$
$\mathcal{S}_{\texttt{screw}}$

Fig. 7: Statechart for a screwdriving skill

applications. Skills are not atomic, as we provide building blocks to users that are on a smaller scale than skills. They are composed of kinematic elements, controllers, monitors, and scripts. While these sub-elements are still programmed by experts using a general-purpose programming language like C++ or Python, they provide a general functionality that can be used to create a large variety of customized skills. Even non-expert users should be able to create their own customized skills by combining the sub-elements. Skills are also extensible. Existing skills can be cloned, refined, and extended by adding new elements that provide additional functionalities. Our skill library comprises multiple layers of skills where each skill can be reused for creating new skills.

While tasks descriptions are mostly independent of the robot in use, some controller parameters, especially for force control, are still highly dependent on the robot and environment. Reducing the need for parameter changes when switching hardware components is part of our ongoing research to further increase reusability.

Acknowledgment

This research has received funding from the European Union's Horizon 2020 Research and Innovation Programme under grant agreement no. 688642, *RAMPup*.

References

[1] T. Hasegawa, T. Suehiro, and K. Takase, "A model-based manipulation system with skill-based execution," *IEEE Transactions on Robotics and Automation*, vol. 8, no. 5, pp. 535–544, 1992.

[2] M. T. Mason, "Compliance and force control for computer controlled manipulators," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 6, pp. 418–432, 1981.

[3] C. Samson, M. Le Borgne, and B. Espiau, *Robot control: The task function approach*. Oxford University Press, 1991.

[4] J. De Schutter, T. De Laet, J. Rutgeerts, W. Decré, R. Smits, E. Aertbeliën, K. Claes, and H. Bruyninckx, "Constraint-based task specification and estimation for sensor-based robot systems in the presence of geometric uncertainty," *International Journal of Robotics Research*, vol. 26, no. 5, pp. 433–455, 2007.

[5] H. Bruyninckx and J. De Schutter, "Specification of force-controlled actions in the task frame formalism - a synthesis," *IEEE Transactions on Robotics and Automation*, vol. 12, no. 4, pp. 581–589, 1996.

[6] M. Klotzbücher, R. Smits, H. Bruyninckx, and J. De Schutter, "Reusable hybrid force-velocity controlled motion specifications with executable domain specific languages," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2011, pp. 4684–4689.

[7] I. Weidauer, D. Kubus, and F. M. Wahl, "A hierarchical extension of manipulation primitives and its integration into a robot control architecture," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2014, pp. 5401–5407.

[8] H. Mosemann and F. M. Wahl, "Automatic decomposition of planned assembly sequences into skill primitives," *IEEE Transactions on Robotics and Automation*, vol. 17, no. 5, pp. 709–718, 2001.

[9] U. Thomas, G. Hirzinger, B. Rumpe, C. Schulze, and A. Wortmann, "A new skill based robot programming language using UML/P statecharts," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2013, pp. 461–466.

[10] I. Kresse and M. Beetz, "Movement-aware action control — integrating symbolic and control-theoretic action execution," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2012, pp. 3245–3251.

[11] D. Vanthienen, M. Klotzbücher, J. De Schutter, T. De Laet, and H. Bruyninckx, "Rapid application development of constrained-based task modelling and execution using domain specific languages," in *IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)*, 2013, pp. 1860–1866.

[12] F. Nägele, L. Halt, P. Tenbrock, and A. Pott, "A prototype-based skill model for specifying robotic assembly tasks," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2018, pp. 558–565.

[13] R. Smits, "Robot skills: Design of a constraint-based methodology and software support," Ph.D. dissertation, Katholieke Universiteit Leuven, Leuven, 2010.

[14] H. Bruyninckx, "Open robot control software: the OROCOS project," in *IEEE International Conference on Robotics and Automation (ICRA)*, 2001, pp. 2523–2528.