RAZER – A Human-Robot Interface for Visual Task-Level Programming and Intuitive Skill Parametrization

Franz Steinmetz[1], Annika Wollschläger[2], and Roman Weitschat[1]

***Abstract*—Maintaining competitiveness and mitigating health issues test test caused by unergonomic working conditions are two main reasons for automating production processes. But such automation is expensive, also because test test experts are required to program the robots. One approach to lowering these costs is to enable shop-floor workers to program robots by providing task-level programming tools. Task-level programming is an established approach, yet appropriate workflows for experts and shop-floor workers remain to be defined. The objective of this paper is to evaluate RAZER, a framework for robot task-level programming, in which skill programming and parameter interface definitions are integrated. The framework provides workflows for both experts – creating skills and providing their parameter interfaces – and for shop-floor workers – using these skills to create executable robot tasks in an intuitive human-robot interface (HRI). The HRI is a graphical user interface that runs in a browser, and provides access to other man-machine-interfaces, such as Programming by Demonstration. Two pilot and two user studies proof that RAZER fulfills the demands of both experts and novice users.**

***Index Terms*—Software, Middleware and Programming Environments, Human-Centered Automation, Intelligent and Flexible Manufacturing, Factory Automation, Human Factors and Human-in-the-Loop**

I. Introduction

AUTOMATING production processes is expensive, not only because of the machines and robots that are required, but mainly because experts are constantly needed to program them. As this is only cost-effective for high batch sizes, *small and medium-sized enterprises* (SME) cannot afford such automation. At the same time, the batch size in manufacturing is also decreasing for large companies, e. g., due to products becoming more and more customized.

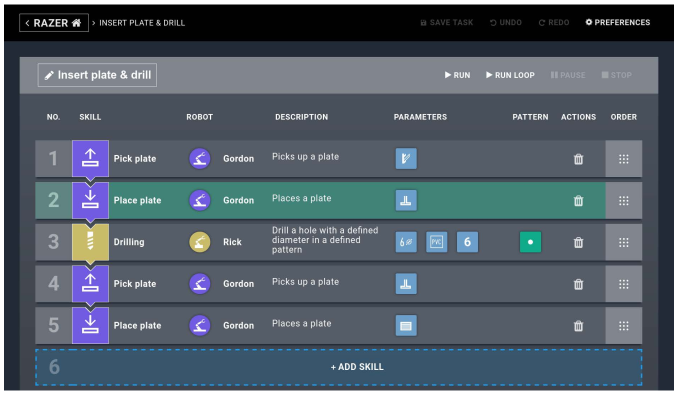
To cost-effectively automate production further, shop-floor workers should be enabled to instruct a robot at their work place. To achieve this, the expertise required to program robot must be lowered drastically. One suggested solution for this is task-level programming with robot skills [1]. It raises the level of abstraction of programming, making it more accessible.

Manuscript received: September 10, 2017; Revised December 8, 2017; Accepted January 7, 2018. This paper was recommended for publication by Editor Tamim Asfour upon evaluation of the Associate Editor and Reviewers' comments. This work has been partially funded by the Helmholtz Association Germany as part of the project RACELab.

[1]Franz Steinmetz and Roman Weitschat are with German Aerospace Center (DLR), Institute of Robotics and Mechatronics, Weßling, Germany. franz.steinmetz@dlr.de

[2]Annika Wollschläger is with Franka Emika GmbH, Munich, Germany. annika.wollschlaeger@franka.de

Digital Object Identifier (DOI): see top of this page.



NO.	SKILL	ROBOT	DESCRIPTION	PARAMETERS	PATTERN	ACTIONS	ORDER
1	Pick plate	Gordon	Picks up a plate				
2	Place plate	Gordon	Places a plate				
3	Drilling	Rick	Drill a hole with a defined diameter in a defined pattern	6⌀ PVC 6	●		
4	Pick plate	Gordon	Picks up a plate				
5	Place plate	Gordon	Places a plate				
6							

Fig. 1. The RAZER HRI runs in a browser. Here, an active drilling task with five skills executed by two robots is shown.

Task-level programming is an established approach, yet appropriate workflows for experts and shop-floor workers remain to be defined. Experts program skills and require a framework that does not place any restrictions on the underlying language or middleware that is used. They need to write complex programs that are generic and error-tolerant. In addition, they have to define the skill parameter interface for shop-floor workers.

In contrast, shop-floor workers typically do not have programming skills. They require a user-friendly *human-robot interface* (HRI) that allows them to easily and quickly sequence skills and parameterize them intuitively, e. g., with the help of *man-machine-interfaces* (MMIs) allowing for kinesthetic teaching. Workers should be able to execute and monitor tasks that they have created.

These requirements have guided the development of *RAZER*, our task-level programming framework for both experts and shop-floor workers. It can be used to create programs for typical industrial tasks in the area of assembly, machine tending, material handling and processing.

The objective of our paper is to develop and evaluate a framework for robot task-level programming. It aims at providing workflows for both experts – creating skills and providing their parameter interface – and for shop-floor workers – using these skills to create executable robot tasks in a intuitive HRI. In addition, we describe parametrization approaches and their tight integration into the framework. Experts can define parametrization procedures and conversion routines. These procedures are followed by the frontend in expressive user dialogs, simplifying the parametrization process for the user, who is confronted only with task-oriented parameters.

II. Related work

*Programming by Demonstration (*PbD) is a well-known approach to reducing both the effort and expertise required for robot programming [2]. In PbD, a robot is taught by an operator via *teleoperation*, *observational learning* or *kinesthetic teaching*. Hereby, complex and previously unknown motions can be shown to a robot once or multiple times. These motions are modeled and often generalized, e. g., by utilizing *Dynamic Movment Primitives (*DMPs) [3].

Task-level programming with robot skills is another approach to facilitate robot programming. An early development into this direction was by Archibald and Petriu [4] and was later refined by Bøgh et al. [5]. The approach uses a three-layered architecture. The lowest level is formed by *device primitives* (sometimes also called *skill primitives*) for simple motion control. The next level is defined by *skills* that are composed of these primitives, combined with logic and sensory input. Skills are parameterizable to tailor them to the task at hand. This parametrization is performed for *tasks*, the highest level, where skills are sequenced. Skills can also have pre- and postconditions, allowing both their applicability and outcome to be checked.

Both of these approaches are combined in this work, exploiting their advantages and reducing their drawbacks. PbD by itself lacks capabilities such as decision making depending on sensory input. Skills on the other hand often require complex parameters. Therefore, we use skills to define the execution logic and PbD for complex parameter specification.

If an operator is to program a robot (so-called *end-user programming*), a HRI is required. The less experienced a user is, the higher are the demands on intuitiveness and usability. Such an interface is usually based on a graphical frontend. From the beginning of task-level programming, an icon-based programming interface has been part of the system [4]. There have been initiatives to establish standards, such as the MORPHA style guide [6]. However, none of these have become widely accepted. Thus, the variety of concepts used by HRIs and robot programming systems is large. An overview is given in [7].

Up to now, a variety of end-user programming frameworks have been developed, which target at users with different levels of expertize. The following incomplete list is approximately sorted by decreasing demands on the user.

The RoboGraph robot programming framework uses Petri nets for the implementation of robot programs [8]. It is shown how complex multi-robot tasks can be handled by the system.

Behavior trees are also used for task-level programming. One example is the HRI of *CoSTAR* [9]. The system also integrates a perception module and interfaces to various components. The official HRI of Baxter, called *Intera*, can also be named here, though information about it can only be found on their website[1].

Many graphical programming interfaces are based on *Scratch* or its concepts [10]–[12]. The system described in [10] allows to define skills either using a *graphical user interface (*GUI) or kinesthetic teaching, but focuses on autonomous skill learning and error detection using its episodic and compositional memory. *Hammer* integrates a 3D environment into the Android-based GUI, which can be used to teach in poses or trajectories [11]. *Code3* lets programmers without experience in robotics first create *CustomLandmarks* and *CustomActions*, before they can be used for manipulation tasks in a drag-and-drop programming interface based on *CustomPrograms* [12].

Simpler interfaces restrict the logical flow to a series of chained skills. In [1], a simple GUI allows for skill sequencing. As part of the HRI, parameters of the skills can be defined using gestures. The HRI for the mobile manipulator AIMM in [13] splits the task specification into a specification and teaching phase. Kinesthetic teaching is used for the parametrization. The authors admit that the GUI is not yet intuitive enough. The system described in [14] focuses on a skill design based on compliant motions, but also provides a thought-out GUI for combining skills and parametrization also using kinesthetic teaching. Advanced users can extend the logical flow using branches and loops.

The frontend of none of these systems reach the usability and intuitiveness of RAZER. In addition, no concept exists on how experts provide skills for novice users.

The parametrization is considered in most of the listed frameworks, whereas its importance is neglected. We have already argued on the relevance of the parametrization process in [15] and have suggested a number of guidelines for the process to be more intuitive and user-friendly. Many of these concepts have been adapted in this work.

III. Entity definition

The fundament of RAZER is a class hierarchy, which we have established for the definition of all available entities (skill, tasks, etc.), shown in Fig. 2. Every class derives from the base class `Entity`. This base class provides interfaces both for *JavaScript Object Notation (*JSON) serialization (required for *inter-process communication (*IPC)) and *object-relational mapping (*ORM) integration (required for database storage). It also defines common attributes, such as `name` and `description`. This class hierarchy, as well as all other components are written in Python, allowing the straight-forward integration of huge variety of other languages, middleware and systems.

`Services` define the interfaces to MMIs. These services are mainly used for the parameterization of skills (see Section IV-E) and are accessed via a *representational state transfer (*REST) interface (see Section IV).

A robot task program is described by a `Task`. A `Task` holds a sequence of skills and a list of robots it requires, i. e., robots that are supposed to execute one of these skills.

Each real-world robot is represented by a `Robot` object. A robot has a list of parameters, specifying for example its number of joints. It also holds links to one or more programs (`libraries`) responsible for the initialization of the robot (see Section IV-B).

Robot `Skills` are the fundament of the system as they represent actions a robot can perform. A skill does not contain the robot program itself, but only a link to the program

[1] http://mfg.rethinkrobotics.com/intera/

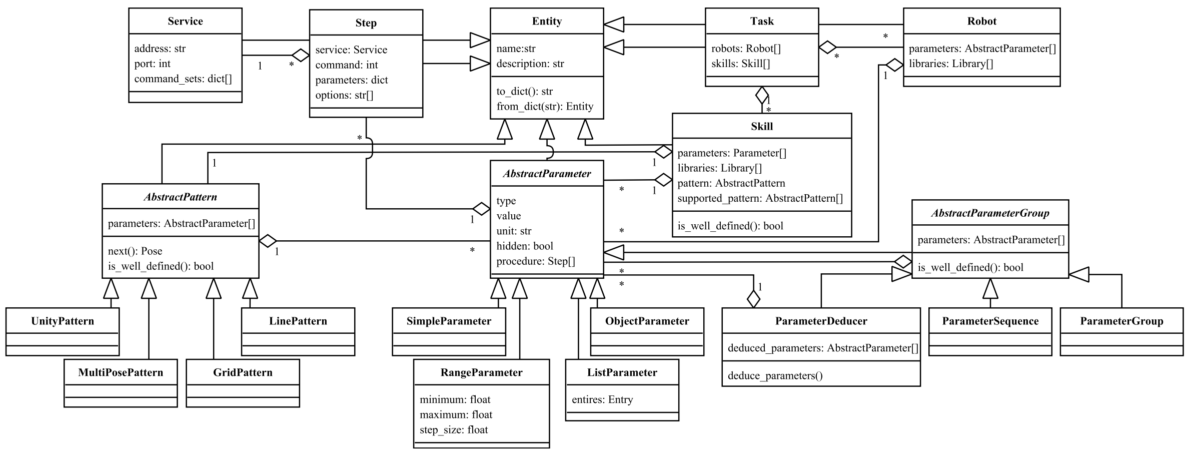


Fig. 2. Simplified class diagram of the RAZER entities, with `Entity` as common base class. Missing classes are for example `Library` or `Trajectory`.

(`libraries`). It does contain the list of available parameters. Some skills also take a pattern as attribute.

An `AbstractPattern` defines a pattern with a single or multiple poses, which can have a geometric shape, such as a grid or a line. Each pattern class must implement a pose generator, calculating for example all grid poses based on three corner poses and the number of rows and columns.

One focus of the system is on the definition of parameters, which can be specified for robots, skills and patterns. There are different types of parameters, such as `SimpleParameter` for base types (`int`, `float`, ...) or `ListParameter` for a selection between predefined options. Parameters always have a `type` and a `value`.

For all parameters, a `procedure` can be defined. A procedure contains a list of `Step`s defining how a parameter value is retrieved. Each step references a `Service` together with a `command` and optional `parameters` for the REST request. A list of `options` specifies what the user can do (e. g., cancel, confirm or just wait).

Parameters can be grouped into a `ParameterGroup` or a `ParameterSequence`. The latter defines an order in which a user has to specify values of these parameters. A further class for grouping parameters is a `ParameterDeducer`. Hereby, subclasses must implement a `deduce_parameters` method. This method allows for converting a set of input parameters (specified by the user) to a set of output parameters (forwarded to the skill implementation), see Section VI-A for examples.

IV. System architecture

This entity system is used throughout the RAZER framework. The framework is kept modular, allowing single components to be easily substituted. Most of the components are implemented as web services in Python. There is no communication between the following components, except for the frontend, which uses REST and WebSockets to share information with all other components.

A. Storage

All information is stored in a central SQL database. This includes the definition of available services, patterns, skills with parameters and tasks created by the user. The information is modeled using the aforementioned entity system and can easily be converted for the usage in a database using the custom ORM system. This *storage* is equipped with a lightweight REST interface, supporting the direct read and write access over a network connection using standardized requests (`GET`, `POST`, etc.) [16].

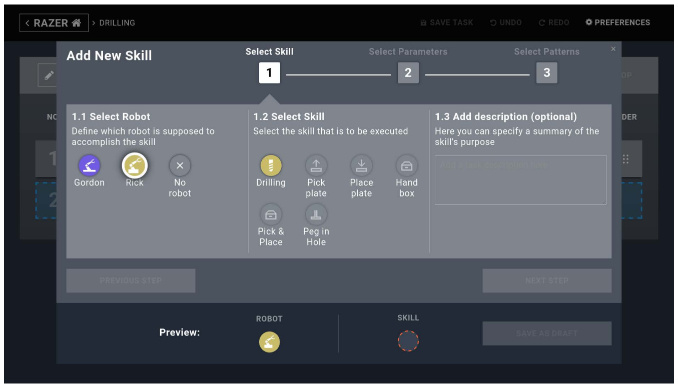
B. Conversion

A task created by a user initially only exists in form of a `Task` object with all its entities. The *conversion* module is responsible for the conversion of such an object into a robot program. A REST interface is used for passing tasks in form of JSON strings that have to be translated. For this, the conversion requires a `Library` for each `Robot` and `Skill`. There can be different kinds of libraries listed in the `libraries` attributes, implemented in different languages. Analogously, one can write different conversion modules.

In our system, a `Library` refers to a RAFCON state machine [17] and also the conversion creates a (wrapping) state machine. First, the state machines for initialization of the robots listed in the `Task` are instantiated and connected with transitions. Next, all state machines of the skills are instantiated and sequenced in their order. If a skill has a pattern, the skill is instantiated once for each pose. The referenced libraries (in our case the state machines) need to accept the parameters defined in the `Skill`. The conversion then passes the user defined values of each skill parameter to the corresponding parameter of the library. For `ParameterDeducers`, the parameters are first converted using its deducer method.

C. Execution

A separate module, the *execution*, is responsible for running a program generated by the conversion. The implementation of the execution therefore also depends on the type of `Library`. It can be commanded again using a REST interface. The interface allows for loading a program and controlling its execution (start, stop, pause, resume). The execution process is constantly observed, the gathered information, e. g., which skill is currently running, is published using a WebSocket and visualized by the frontend.

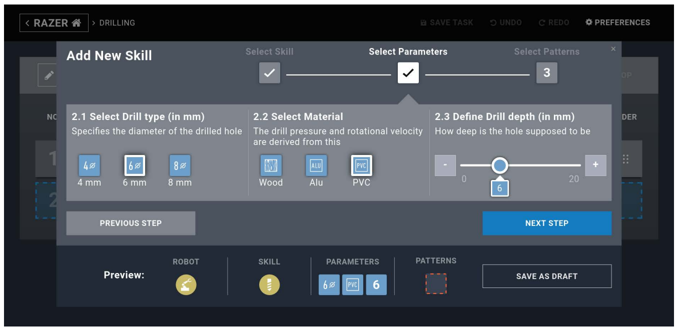


1.1 Select Robot
Define which robot is supposed to accomplish the skill

1.2 Select Skill
Select the skill that is to be executed

1.3 Add description (optional)
Here you can specify a summary of the skill's purpose

Fig. 3. The first steps when creating a new skill is choosing the robot, selecting the skill and optionally changing the description. Skills unsupported for the selected robot are greyed out.



2.1 Select Drill type (in mm)
Specifies the diameter of the drilled hole

2.2 Select Material
The drill pressure and rotational velocity are derived from this

2.3 Define Drill depth (in mm)
How deep is the hole supposed to be

Fig. 4. Having selected a skill, the user sets the desired parameters. The two parameters on the left hand side are of type `ListParameter`, the right one of type `RangeParameter`.

D. Frontend

The central module is the *frontend*. It is a graphical HRI, based on web technologies (HTML, CSS [Foundation], JavaScript [AngularJS]) and thus runs on all modern browsers and a variety of devices (PC, tablet, smartphone, . . .). The interface was designed in cooperation with a professional interface designer, targeting mainly at non-expert users such as shop-floor workers.

The GUI supports the task creation and skill parametrization process by offering wizards. The skills are sequenced and can be reordered using drag and drop. The screenshots in Figs. 1, 3 and 4 show some example views, Section V-B gives more details about the workflow.

E. Services

Services act as connection to arbitrary (external) *man-machine-interfaces (*MMIs), such as a robot, or other components, such as a world model. Services can be commanded to *start* and *stop* (e. g., the gravity compensation mode), to execute an *action* (e. g., closing the gripper) or to *retrieve* a value (e. g., the robot's pose) which is subsequently used as parameter value.

V. Workflow

In this section, we will show the common workflow with the system. This workflow depends on whether the operator is an expert/skill provider or a non-expert/user.

A. Expert's perspective

Experts are responsible for providing parameterizable skills to the user. If they choose RAFCON for the implementation of these skills, there is already a conversion and execution module. If not, they have to provide specific modules for their skill language. Alternatively, they can wrap their implementation into a RAFCON state machine, as presented in Section VI-B.

The RAZER architecture does not impose constraints on how a skill should be implemented. Ideally, a skill is both generic and robust. It should be parameterizable so it can be adapted to specific tasks.

Having created a robot skill, the expert has to define the interface to the skill by creating a `Skill` object. Hereby, its name, description and optional icon for the skill are determined and the skill program is assigned using a `Library`.

The specification of the parameter interface is of importance, as it affects the workflow of the end-user. First of all, the name and type of each parameter must be stated. Often, it is helpful for the user if a plausible range for the parameter value is given (`RangeParameter` with minimum, maximum and step size) or if one can choose amongst certain options (`ListParameter`). An example on how this is visualized in the user interface is given in Fig. 4. The *Secondary parameters* [15] should be set to `hidden`. They can still be accessed in the frontend in an advanced mode, but are not mandatory to be specified. If parameters of a skill are difficult to define, but can be derived automatically from more intuitive task-level parameters, a `ParameterDeducer` should be used. As mentioned in Section III, procedures can be assigned to parameters. A procedure defines which steps are needed to retrieve a parameter value. For each step, a service is called, for example to open a gripper. Section VI gives some examples.

The choice of patterns must also be considered. The majority of skills needs to know where an action has to be performed. Furthermore, these actions are often repetitive, i. e., they need to be done several times with changing locations (e. g., multiple drilling locations). Instead of requiring the user to add and parameterize a skill for each of these locations/poses, a skill can provide options for passing a pattern consisting of multiple poses to a single skill. For this, there are a number of predefined patterns available, such as `LinePattern` for poses in a row, `GridPattern` for poses forming a grid or `MultiPosePattern` for arbitrary distributed poses.

Finally, the expert has to setup the required services. Services are mainly used within procedure steps of parameters, where MMIs should be used for the parametrization. The service used most in our system is the *gravity compensation service*, which allows users to hand-guide the robot and query its current pose.

B. User's perspective

Users of the system are normally non-experts that want to define tasks for robots. They only interact with the HRI (frontend) that is typically running on a tablet (see accompanying video). When opening the HRI, the user can choose between opening a previous task or creating a new one. A task is then presented on a new view as a vertical sequence of skills, see Fig. 1. Also the chosen parameter values and the pattern type are visualized for each skill.

The user can add a new skill by clicking a button at the end of the skill list. A wizard opens and guides the user through the skill creation process. The first step in this wizard is to select the robot that is to perform a certain skill (Fig. 3). This selection updates the list of available skills, deactivating skills not supported by the robot. After that, the user chooses a skill and can optionally adjust the description.

In the next step, the values for the parameters are specified (Fig. 4). All non-hidden parameters are shown with name and inputs: For a `RangeParameter`, a slider allows for choosing a value, for a `ListParameter`, several buttons with icons are displayed. Parameters with a procedure are handled specially. For those, a button "Start teaching" is shown, leading the user in a separate wizard step by step through the procedure. Hereby, a dialog shows information about the current step and what the user is supposed to do. In a `retrieve` step, the parameter value is determined by querying the assigned service. When all parameter values are specified, a further button click appends the skill to the task.

At any time, the user can alter the current task. Skills can be deleted or rearranged via drag and drop. Their parameter values can be changed independently without having to repeat the whole wizard again, just by selecting them. A history allows for undo and redo of operations.

Having finished the task, the user only needs to click the play button to start the execution. The task is automatically uploaded, converted and loaded by the execution. During the actual execution, the active skill is highlighted. The execution can also be stopped, paused and resumed by the user with the buttons in the execution bar. Skills can make use of a special service, allowing the user to be queried for information. This is for example used to show a dialog after a collision, asking whether to continue or abort.

VI. Pilot studies evaluating expert's perspective

Up to now, we have successfully employed RAZER in two projects. They are presented in the following section and the accompanying video, providing several examples for the various features of the system.

A. RACELab

RAZER has originally been developed for our RACELab project, in which intuitive robot programming was one major aim. The hardware setup consists of two KUKA LWR 4+ robots (see Fig. 5a), of which one ("Rick") is equipped with a custom driller, the other one ("Gordon") with a Robotiq gripper and mounted on a linear axis for extended reachability. Behind the workbench, there is a shelf with plates, which Gordon can grab. Next to Rick, there is a fixture for a single plate with a drilling stencil for a number of holes. For the implementation of skills, we use hierarchical state machines created with RAFCON [17], for which our conversion and execution modules are specifically designed.

In one scenario, Gordon is supposed to grasp a plate from the shelf and put it into the fixture. Consequently, the task of Rick is to drill holes, which have previously been defined by the human instructor. Finally, Gordon picks the plate again and stacks it on a storage.

For this purpose, there are two skills for Gordon, "Pick plate" and "Place plate" with `ListParameters` "From" and "Target", respectively. The "Drilling" skill for Rick is more sophisticated and implements the programming scheme suggested in [15]. The `ListParameters` "material" and "hole diameter" and the `RangeParameter` "hole depth" currently only serve an illustrative purpose, but could later be used in combination with a `ParameterDeducer` to extract the optimal rotational speed and contact pressure.

For a second scenario, we developed "Pick & Place" and "Peg in hole" skills for Gordon. The skill implementation of "Pick & Place" requires seven parameters with lists of poses ("pre pick poses", "post place poses", etc.) for a via-point interpolator. As these parameters would be too tedious for a user to teach, one is only confronted with three parameters "Pick trajectory", "Place trajectory" and "Depart trajectory". These three parameters use the *trajecotry recorder service* in their procedure for value retrieval. They are nested in a `ParameterSequence` (defining that they are taught in a row), which itself is nested in a `ParameterDeducer` (converting the three trajectories into a few feature via-points for each of the seven parameters). To give an example for a procedure, the steps for the pick trajectory are: (1) "open gripper", (2) "start recording" (starts gravity compensation mode), (3) "instruct user and wait for confirmation", (4) "stop recording", (5) "retrieve trajectory" (stops gravity compensation mode) and (6) "close gripper".

In the HRI, this complexity is hidden from the user. One only clicks on "Start teaching" and follows the instructions. The gripper is opened automatically, then the user is asked to move the gripper to the pick position (with the arm in gravity compensation mode) and to confirm this consequently. The place and depart trajectory are recorded analogously.

B. SMErobotics

A second example, for which RAZER is used, is a demonstration for the automobile manufacturer Daimler. The robots are supposed to fasten screws in a part of a motor block as instructed by a user.

The hardware and software infrastructure were based on the SMErobotics environment, described in an earlier version in [18]. There are two KUKA iiwa robot arms mounted on a table with the work-cell containing the motor block in between (see Fig. 5b). Both robots have a parallel gripper and share a common electric screwdriver, which they are able to pick up and run autonomously. The robots can be commanded using the Robotics API with Java. Thus, the skills have been implemented in Java. In order not having to create custom conversion and execution modules, each skill was wrapped in a RAFCON state machine.

The "Screwing" skill picks up a screw, inserts it into the taught hole and fastens it. The screw storage is determined

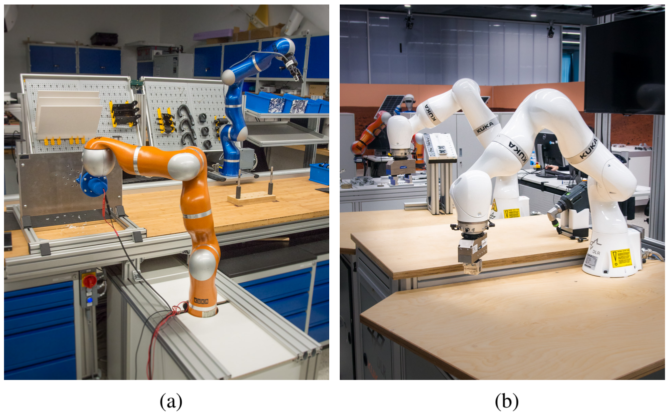


Fig. 5. Overview over the hardware setups of the two projects RACELab (a) and SMErobotics (b). *Rick* is the orange and *Gordon* the blue LWR, *Diiwa* (back) and *Iiwan* (front) are the two white iiwa robots.

by a `ListParameter` "screw type". For the hole, a number of patterns are supported. The procedure of pattern make use of a special *skill execution service*. They for example include a "pick up screwdriver" step at the beginning and a "place screwdriver" step at the end. These ensure, that the screwdriver is automatically picked up by the robot before the teaching process and returned thereafter.

There is also a special skill not assigned to any robot. It allows to prompt a user dialog in the frontend. We use it as first skill within a task to query the user at the beginning of the execution to check whether everything is setup correctly.

C. Discussion

Despite the hardware similarities between the two setups, the software stack beneath is completely different. Instead of wrapping the Java programs into state machines, we could alternatively write custom *conversion* and *execution* components for other programming languages than Python. As a robot expert, it is helpful not being restricted to any specific programming language. For both projects, the required services could be written without greater effort and reused in different skills.

Furthermore, the skill parameter interface can be defined with various tools. Having programmed a skill, the expert is required to invest some additional time to provide an appropriate parameter interface for the end user. For this, the expert is given a number of powerful tools, allowing for example the integration of any MMI (using services), guidance of novice users through the parametrization (using procedures) or conversion of parameters to be task-oriented (using `ParameterDeducers`). There is no task-programming framework that offers similar properties.

VII. User Studies Evaluating User's Perspective

The frontend of RAZER was evaluated in two user studies to assess the usability and intuitiveness of the GUI.

A. Procedures

1) Cognitive Walkthrough: The first study was conducted as a *Cognitive Walkthrough* (CW) [19], [20] with only two

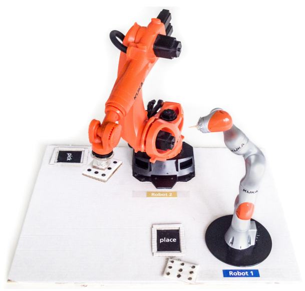


Fig. 6. For the thinking aloud user study, this miniature work station with two flexible toy robots was used.

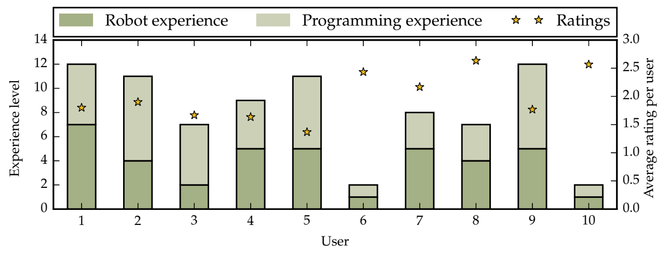


Fig. 7. The experience with robots and programming of the ten participants of the thinking aloud study is compared to their average rating.

participants, both robot experts. In an introduction, the participants were put into the context of the intended target user group: A text described the task scenario and a video showed a shop-floor worker assembling an auxiliary heating unit fully manually in a company. The purpose of RAZER is letting non-expert workers instruct robots to support them in their daily work. Besides this contextual information, the users did not receive further instructions into the software.

They were given four tasks (see Section VII-B) with a detailed step by step guide on which actions to perform in order to solve the tasks. After each step performed, the participants were asked to fill in the CW protocol, asking about the *conformity with user expectations* (I), *visibility of the input* (II), *recognition of the input* (III) and the *comprehensibility of the feedback* (IV). They were also asked to speak out loud every thought that came into their mind.

Having accomplished all tasks, the users were given the ISONORM 9241/110 questionnaire [21] in its long form, which is intended to evaluate software regarding the international usability standard ISO 9241, part 110. Hereby, a software is rated by the seven software requirements *suitability for the task*, *self descriptiveness*, *conformity with user expectations*, *suitability for learning*, *controllability*, *error tolerance* and *suitability for individualization*. The last requirement was left out in the questionnaires, as it could not be judged by purely fulfilling the given tasks. Each requirement is concretized in six descriptions, for which the user gives a score between -3 (worst) and $+3$ (best) on a Likert scale with seven levels.

After a glance over the test results, mainly concerning the oral and written feedback given, the RAZER frontend was improved in many details.

2) Thinking aloud: With the improved version of RAZER, a second user study was carried out. Hereby, $n = 10$ participants evaluated the usability in a user study based on *thinking aloud* tests as described in [22]. None of these users had seen the software before, but all were experienced with touch

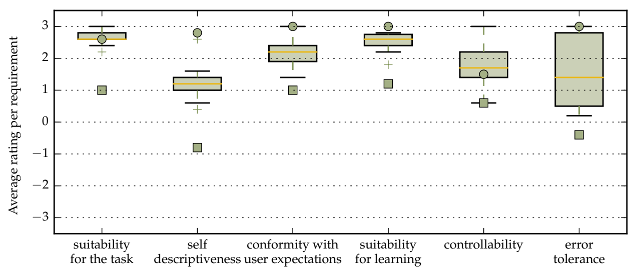


Fig. 8. The plot shows the average rating per requirement. Hereby, the small circles and squares belong to the two users of the CW. The boxplots are the result from the thinking aloud study.

devices (between 4 and 7 on a Likert scale from 1 to 7). The experience with robots and programming in general, estimated by the users again on the same Likert scale, is shown in Fig. 7. The users were given the same introduction and tasks as in the first study. Instead of using real robots for PbD, a miniature work station with two model robots was provided (see Fig. 6). For the examiner, all tasks were split in subtasks consisting of 1–11 actions (typically clicks). Although the users should think aloud all the time, they were also encouraged to do so by the examiner after each subtasks. Subsequently, the participants again filled in the ISONORM 9241/110 questionnaire. Due to the fact that this is a German questionnaire and not all users were German native-speakers, all texts were translated into English.

B. Tasks

The four tasks given to the users were designed to both cover most features of the interface and at the same time reflect typical workflows of a shop-floor worker.

1) Creation of a new task with a *Pick & Place* skill (included teaching of three trajectories using PbD).
2) Addition of *Drilling* skill (required three parameters plus the definition of a rectangular grid of holes using a pattern and PbD) and saving.
3) Modification of a parameter value, saving and execution of the task.
4) Duplication and renaming of the previous task, addition of a user skill in between the existing ones (required switch to home screen and skill rearrangement).

C. Results

1) Cognitive Walkthrough: The CW's purpose was mainly the gathering of qualitative feedback rather than quantitative figures. Nevertheless, the average rating per requirement of the ISONORM questionnaire have been plotted for each of the two users in Fig. 8 with circles and squares. Except for the self descriptiveness, the average requirement rating is above 1, whereas the total average is 1.4.

The written and oral feedback was often very detailed and generally helpful. The participants found inconsistencies in the design, missed descriptions in dialogs and had issues with some labels or the choices of colors (e. g., an orange robot icon was interpreted as warning). Also some features were desired, such as buttons to clear input fields.

Despite the critics, the participants agreed for the 82 actions that had to be performed in total that criterion I was fulfilled in 84% of the cases, 87% for II, 91% for II and 83% for IV.

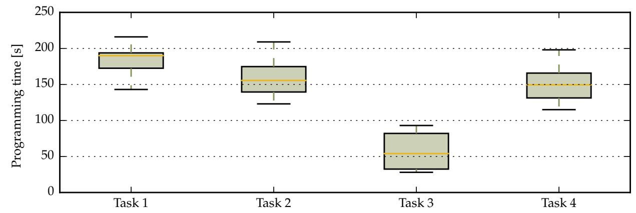


Fig. 9. The boxplot visualizes the programming time required for each of the four tasks by the participants in the thinking aloud study.

2) Thinking aloud: The average rating per user is presented in Fig. 7, the average rating per ISONORM requirement is given in the boxplot in Fig. 8. The overall average is 2.0. Except for the error tolerance, the *interquartile range (*IQR) is 0.8 or below. The programming time for the four tasks is displayed in the boxplot in Fig. 9.

Besides the rating from the questionnaire, a lot of qualitative feedback was collected. All except one user solved all tasks without any external help. Nevertheless, three main issues could be identified, which some users struggled with: (i) Modification of a parameter value (they first clicked on the skill icon in the task view instead of the parameter icon), (ii) teaching of trajectories (they were not fully sure about the given instructions such as "move gripper to desired object"), and (iii) definition of a selected pattern (they were unsure about the orientation of the grid for descriptions such as "upper left corner of the grid"). Overall, most users were surprised about the usability and described the software as intuitive and natural to operate.

D. Discussion

The feedback of the CW was used to further improve the usability of the RAZER HRI. The increase of the average rating from 1.4 to 2.0 shows that this was successfully accomplished.

The current score assesses RAZER a remarkable usability, as 2.0 is a value higher than any of the 41 software programs evaluated with the same questionnaire in [23]. The significance of this value is supported by the fact that the IQR is 0.8 or below for all requirements except the error tolerance. The correlation between the user experience and their rating is -0.73, indicating that the software is perceived to be even more user-friendly for non-expert users, for which the interface targets at. The feature coverage of the tasks and the ratings confirm that the RAZER frontend is a highly user-friendly and intuitive HRI, especially for novice users.

The programming times of less than three minutes per skill (task 1, 2 and 4) indicate that users of any experience can quickly command the robot with the HRI. Having created a task, it can even faster be adapted to a changed parameter (task 3). Considering that the participants had neither seen the interface before nor were given any instructions, the programming time will probably decrease with continued software usage.

Despite the previous ignorance of the software, only one time external help was needed. Nonetheless, the second study still uncovered three issues in the usability. For issue (i), a simple instruction into the software would have already helped. For (ii) and (iii), the users suggested to add graphical support, such as an image, demonstrating the needed trajectories and a pattern with annotations for the required teaching locations. Furthermore, some dialogs could be more self descriptive or show explanations on demand. The perceived error tolerance could probably be improved by fixing some issues that were revealed during the study. All these changes can be implemented without greater effort.

Only a few other HRI mentioned in Section II have been evaluated in a user study, making it difficult to compare them. The study in [12] about the HRI using Scratch demonstrates that despite intense training, novice user failed to solve 50% of the tasks and rated the interface as marginally acceptable. Behavior trees also provide great flexibility in terms of logical flow. Yet, this comes at cost of intuitiveness and usability, with programming tasks of over seven minutes for a simple task. Almost half of the users in a study about CoSTAR state they would need expert help to use such an interface [24]. The study also reveals that unexperienced users have problems with the concept of frames. The HRI of AIMM shares similarities in the feature set of the RAZER frontend. Even though, people with low experience level required support from the supervising expert even for the task of adding and parameterizing a single pick and place skill [13]. In addition, they took over five minutes to solve the task. The approach using gestures is restricted in the number of possible skills and their types of parameters [1].

RAZER only allows a sequential order of skills. We do not see this as a big restriction, as most workflows in the targeted industrial domains (assembly, machine tending, material handling and processing) are also sequential. When looking at the industrial-related tasks given to the users in the studies referenced in the previous paragraph, they are all pure sequences. Of course, branching and loops are for example required for error handling, but all this is hidden from the user within a skill. In addition, the system allows for loops in form of patterns (repeats the skill for every hole) and by running the whole task in a loop.

For the future, we want to further extend the framework. The system should, for example, allow for the execution of single skills within a task, which improves the testing of new tasks. Another essential enhancement will be the integration of pre- and postconditions of skills. This will allow the verification of tasks before their execution.

VIII. Conclusions

The presented software framework for task-level programming, called RAZER, is a highly suitable tool for both robotic experts – providing skills and their interface – and shop-floor workers – creating tasks from parameterized skills. Experts are not restricted in their software and have at the same time a powerful toolset to define skill parameter interfaces. Novice users can easily program tasks and parameterize skills with the help of the confirmed intuitive and user-friendly *human-robot interface (*HRI) with integrated *Programming by Demonstration (*PbD).

Acknowledgment

The authors would like to thank B. Weber, T. Wehrle and D. Janetzko for their valuable help in conducting the user studies.

References

[1] M. Pedersen, D. Herzog, and V. Krüger, "Intuitive skill-level programming of industrial handling tasks on a mobile manipulator," in *IEEE Int. Conf. Intelligent Robots and Systems*, 2014, pp. 4523–4530.

[2] A. Billard, S. Calinon, R. Dillmann, and S. Schaal, "Robot programming by demonstration," in *Springer handbook of robotics*. Springer, 2008, pp. 1371–1394.

[3] A. J. Ijspeert, J. Nakanishi, H. Hoffmann, P. Pastor, and S. Schaal, "Dynamical Movement Primitives: Learning attractor models for motor behaviors," *Neural computation*, vol. 25, no. 2, pp. 328–373, 2013.

[4] C. Archibald and E. Petriu, "Skills-oriented robot programming," in *Proc. Int. Conf. Intelligent Autonomous Systems*, 1993, pp. 104–15.

[5] S. Bøgh, O. S. Nielsen, M. R. Pedersen, V. Krüger, and O. Madsen, "Does your robot have skills?" in *43rd Int. Symp. on Robotics*, 2012.

[6] R. Bischoff, A. Kazi, and M. Seyfarth, "The MORPHA style guide for icon-based programming," in *Proc. 11th IEEE Int. Workshop Robot and Human Interactive Communication*, Sept. 2002, pp. 482–487.

[7] G. F. Rossano, C. Martinez, M. Hedelind, S. Murphy, and T. A. Fuhlbrigge, "Easy robot programming concepts: An industrial perspective," in *IEEE Int. Conf. Automation Science and Engineering*, Aug. 2013, pp. 1119–1126.

[8] J. López, D. Pérez, and E. Zalama, "A framework for building mobile single and multi-robot applications," *Robotics and Autonomous Systems*, vol. 59, no. 3, pp. 151–162, Mar. 2011.

[9] C. Paxton, A. Hundt, F. Jonathan, K. Guerin, and G. D. Hager, "CoSTAR: Instructing collaborative robots with behavior trees and vision," in *IEEE Int. Conf. Robotics and Automation*, 2017, pp. 564–571.

[10] S. Hangl, A. Mennel, and J. Piater, "A novel skill-based programming paradigm based on autonomous playing and skill-centric testing," *arXiv preprint arXiv:1709.06049*, 2017.

[11] C. Mateo, A. Brunete, E. Gambao, and M. Hernando, "Hammer: An android based application for end-user industrial robot programming," in *IEEE/ASME Int. Conf. Mechatronic and Embedded Systems and Applications*, 2014.

[12] J. Huang and M. Cakmak, "Code3: A system for end-to-end programming of mobile manipulator robots for novices and experts," in *Proc. ACM/IEEE Int. Conf. Human-Robot Interaction*, 2017, pp. 453–462.

[13] C. Schou, J. Damgaard, S. Bøgh, and O. Madsen, "Human-robot interface for instructing industrial tasks using kinesthetic teaching," in *44th Int. Symp. Robotics*, Oct. 2013.

[14] F. Dai, A. Wahrburg, B. Matthias, and H. Ding, "Robot assembly skills based on compliant motion," in *Proc. 47th Int. Symp. Robotics*, 2016.

[15] F. Steinmetz and R. Weitschat, "Skill parametrization approaches and skill architecture for human-robot interaction," in *IEEE Int. Conf. Automation Science and Engineering*, 2016, pp. 280–285.

[16] Z. Shelby, "Embedded web services," *IEEE Wireless Communications*, vol. 17, no. 6, pp. 52–57, Dec. 2010.

[17] S. Brunner, F. Steinmetz, R. Belder, and A. Dömel, "RAFCON: A graphical tool for engineering complex, robotic tasks," in *IEEE/RSJ Int. Conf. Intelligent Robots and Systems*, Oct. 2016, pp. 3283–3290.

[18] K. Nottensteiner, T. Bodenmüller, M. Kaßecker, M. A. Roa, A. Stemmer, T. Stouraitis, D. Seidel, and U. Thomas, "A complete automated chain for flexible assembly using recognition, planning and sensor-based execution," in *Proc. 47th Int. Symp. Robotics*, 2016.

[19] P. G. Polson, C. Lewis, J. Rieman, and C. Wharton, "Cognitive walkthroughs: a method for theory-based evaluation of user interfaces," *Int. Journal of Man-Machine Studies*, vol. 36, no. 5, pp. 741–773, 1992.

[20] C. Wharton, J. Rieman, C. Lewis, and P. Polson, "The cognitive walkthrough method: A practitioner's guide," in *Usability inspection methods*. John Wiley & Sons, Inc., 1994, pp. 105–140.

[21] K. Figl, "ISONORM 9241/10 und Isometrics: Usability-Fragebögen im Vergleich." in *Mensch & Computer*, vol. 9, 2009, pp. 143–152.

[22] K. A. Ericsson and H. A. Simon, "Protocol analysis: Verbal reports as data," *Psychological review*, vol. 87, no. 3, p. 215, 1980.

[23] J. Prümper, "Der Benutzungsfragebogen ISONORM 9241/10: Ergebnisse zur Reliabilität und Validität," *Software-Ergonomie '97-Usability Engineering: Integration von Mensch-Computer-Interaktion und Software-Entwicklung*, pp. 253–262, 1997.

[24] C. Paxton, F. Jonathan, A. Hundt, B. Mutlu, and G. D. Hager, "User experience of the coSTAR system for instruction of collaborative robots," *ArXiv e-prints*, Mar. 2017.