A novel Skill-based Programming Paradigm based on Autonomous Playing and Skill-centric Testing

Simon Hangl, Andreas Mennel and Justus Piater*

***Abstract*— We introduce a novel paradigm for robot programming with which we aim to make robot programming more accessible for unexperienced users. In order to do so we incorporate two major components in one single framework: *autonomous skill acquisition by robotic playing* and *visual programming*. Simple robot program skeletons solving a task for one specific situation, so-called *basic behaviours*, are provided by the user. The robot then learns how to solve the same task in many different situations by *autonomous playing* which reduces the barrier for unexperienced robot programmers. Programmers can use a mix of *visual programming* and *kinesthetic teaching* in order to provide these simple program skeletons. The robot program can be implemented interactively by programming parts with visual programming and kinesthetic teaching. We further integrate work on experience-based skill-centric robot software testing which enables the user to continuously test implemented skills without having to deal with the details of specific components.**

I. Introduction

Despite large progress in the community to make robot programming more accessible to a larger public, it is still very hard for beginners to get started. Software projects in robotics consist of a complex interplay of many components such as robot control, object detection, machine learning, path planning and many more. Additionally, skill acquisition in robotics is a very hard problem which is still not solved yet. Due to the diversity of the state spaces and the high number of the controlled degrees of freedom it is very hard to design well-generalising controllers. This problem is often solved by introducing large task-specific priors designed by engineers who are experts in a certain subfield.

This is clearly not an option if the goal is to enable beginners to design simple robotic skills. Another possibility is to design abstract visual programming languages with which the users can easily program a skill for a limited range of situations. Provided with such a simple program, the robot can add the rest, e.g. the generality by autonomous playing.

In previous work work we introduced an approach for (semi-) autonomous skill acquisition in which programmers provide so-called *basic behaviours*, that solve a task in a narrow range of situations [1], [2]. This range is extended by autonomous playing, where sequences of behaviours are used to prepare the environment such that the basic behaviour can be applied successfully. We further introduced a novel approach for autonomous skill-centric testing which enables even non-experts to automatically run tests on robot software without the need for designing unit tests [3].

*All authors are with the Department of Informatics, University of Innsbruck, Austria first.last@uibk.ac.at

We propose a novel programming paradigm based on our previous work on autonomous robots [1]–[3]. We extend our system to support visual programming. Developers can implement basic behaviours via drag and drop for a narrow range of situations. Further, developers can interactively add parts of the basic behaviour by kinesthetic teaching. After providing the basic behaviour the rest, i.e. how to generalise to more situations, can be learned autonomously by the robot.

Further, we embedded autonomous testing to our programming framework in which buggy functions can be identified without deep knowledge of robotic systems. This can help to pinpoint problems which can be fixed either by the developers themselves if they have the required knowledge, or by experts that can be consulted based on the output.

II. Contribution

Our contribution is an integrated platform for robot development based on technology for autonomous robotics. We combine previous work on autonomous skill acquisition and robot software testing [1]–[3] in a unified framework. We further extend our framework by a visual programming capability in which a basic behaviours can be implemented in a simple way. Skills implemented this way can then be tested autonomously without requiring the user to be an expert.

We provide an abstract architecture and an open source implementation[1]. We demonstrate that our approach can be used to program simple robotic skills by visual programming and to make them generalise by autonomous robotic playing. This simplifies the implementation process and makes robot programming accessible to non-experts and even end-users.

III. Related work

Our framework aims to contribute to the field of end-user development [4]. Domain-specific end-user programming is a powerful tool in order to enable users to develop applications for their own purpose. Eco-systems for end-user programming have to fulfill specific properties to ensure that end-users are able to understand them properly [5]. Frameworks and programming languages for end-users need to be simple but powerful enough to implement useful programs. One way is to follow a visual programming paradigm using abstract symbols in which the symbols are tailored to the specific domain. Popular examples are the *scratch* framework [6], *Snap!*[2] or *Blockly*[3].

[1] https://github.com/shangl/kukadu
[2] http://snap.berkeley.edu/
[3] https://developers.google.com/blockly/

Visual programming received some attention in robotics in recent years [7]–[9] and it was shown that properly designed visual programming interfaces can make robot programming even accessible to children [10]. Each of these methods incorporates certain assumptions in order to ensure simplicity and the required generality. Jackson published on the Microsoft robotics studio (MSRS) [8] which aims to be a general purpose robotics platform. Similar to the software presented in this paper, interfaces for hardware are defined in order to provide an abstraction from the raw hardware. This hardware can be used in a webservice-like structure, where distributed webservices implement certain skills and behaviours. Similar to our approach, skills and behaviours can be coordinated by visual programming. Kim and Jeon designed a visual programming language based on LabVIEW[4] and the MSRS for Lego mindstorm robots [7].

Nguyen et al. [9] follow a different approach with their system called *ROS Commander* (ROSCo). Generic parametrised skills are developed by experts in the respective field. End-users can combine them by visual programming in the form of hierarchical finite state machines (HFSM). These state machines are a generic representation of a skill which then can be deployed. Other users can load these state machines and adapt the skill to their local environment, e.g. by teaching where certain behaviours should be applied to. Our system uses a mixed paradigm in which the generic skills are either created by the end-user by visual programming and subsequent autonomous playing or by field experts which make the corresponding controller available.

Alexandrova et al. designed a flow-based visual programming language called *RoboFlow* [11]. Skills are either created top-down, i.e. by implementing a skill from scratch by drag & drop, or bottom-up, i.e. demonstrating the solution for one situation and editing it afterwards in a graphical interface [12]. Our system basically follows the top-down approach, however, the user can add parts of the skill by kinesthetic teaching. These parts can be refined by policy reinforcement learning [13]–[15].

Similarly, Wächter et al. introduced the *ArmarX* framework which is provides an abstract view on robot programs following at a statechart concept. Just as our framework, *ArmarX* also follows the idea of an integrated software suite based on C++ in which skill hierarchies can be created.

All these works [7]–[9], [11], [12] use visual programming for skills programming. The key distinction of our approach is the integrated system for autonomous skill acquisition. In our system, the user can provide simple basic behaviours solving a task in on specific situation. The rest, i.e. generalisation to different situations, is then learned autonomously by robotic playing. Further, none of these systems provides an integrated autonomous testing framework.

IV. A SKILL-BASED PROGRAMMING PARADIGM

In this section we sketch our skill acquisition paradigm based on technology for developmental robotics. We provide a formal definition of behaviours and skills by unifying problem formulations from previous work [1]–[3], c.f. section IV-A, and give a high-level description of our framework, c.f. section IV-C. Sections V and VI sketch concrete solutions to the problem definitions given in this section.

A. Behaviours and skills

A *behaviour* $b : S \to S, s \mapsto s'$ is a function that changes the state $s \in S$ of an environment to some other state $s' \in S$. The (partially unknown) states s, s' consist of both, the external state of the environment and the internal state of the robot. Behaviours are basically everything a robot could potentially do, from low-level behaviours such as simple point to point movements to high-level behaviours such as grasping. Behaviours do not come with a notion of a goal or task-specific success. For this purpose we introduce a *skill* $\sigma = (b^\sigma, \text{Success}^\sigma)$, which comes with a success predicate $\text{Success}^\sigma(s)$. It defines whether or not the state $s \in S$ is a target state. A skill execution is successful, if the behaviour b^σ executed in state $s \in S$ transforms the system such that

$$\text{Success}^\sigma\left(b^\sigma(s)\right) = \text{true} \tag{1}$$

The set $D^\sigma = \{s \,|\, \text{Success}^\sigma(b^\sigma(s)) = \text{true}\}$ is called the *domain of applicability* (DoA) of the skill σ. In this paper, the domain of applicability is increased by preparing the environment such that the basic behaviour can be executed to solve the task. This is done by executing behaviour sequences of length k with the property

$$\text{Success}^\sigma\left(b_k \circ \ldots \circ b_1 \circ b^\sigma(s')\right) = \text{true} \tag{2}$$

such that $s' \in S$ was not yet in the DoA of the skill with $s' \notin D^\sigma$. The success predicate can either be given from the outside, e.g. by a human teacher, or by training a sensor model of successful executions.

B. Sensor data and function call profiles

An essential part of the proposed programming paradigm is the storage of experience gathered throughout the lifetime of the robot. Whenever a skill or behaviour is executed, two matrices containing *sensor data* and *function call profiles* are stored. The sensor data per execution of the skill σ with duration T is given by

$$\mathbf{M}_\sigma(\mathbf{s}) = \begin{matrix} \\ \\ \\ \\ t=1 \end{matrix} \begin{matrix} \begin{pmatrix} \mathbf{m}_1(0) & \ldots & \mathbf{m}_1(T) \\ \mathbf{m}_2(0) & \ldots & \mathbf{m}_2(T) \\ \vdots & \ddots & \vdots \\ \mathbf{m}_M(0) & \ldots & \mathbf{m}_M(T) \end{pmatrix} \\ \xrightarrow{\Delta t} \end{matrix} \begin{matrix} \text{sensor } 1 \\ \text{sensor } 2 \\ \downarrow \\ \text{sensor } M \\ t=T \end{matrix} \tag{3}$$

The function call profile matrix stores information on how many instances of all functions are active over the course of

[4] http://www.ni.com/de-at/shop/labview.html

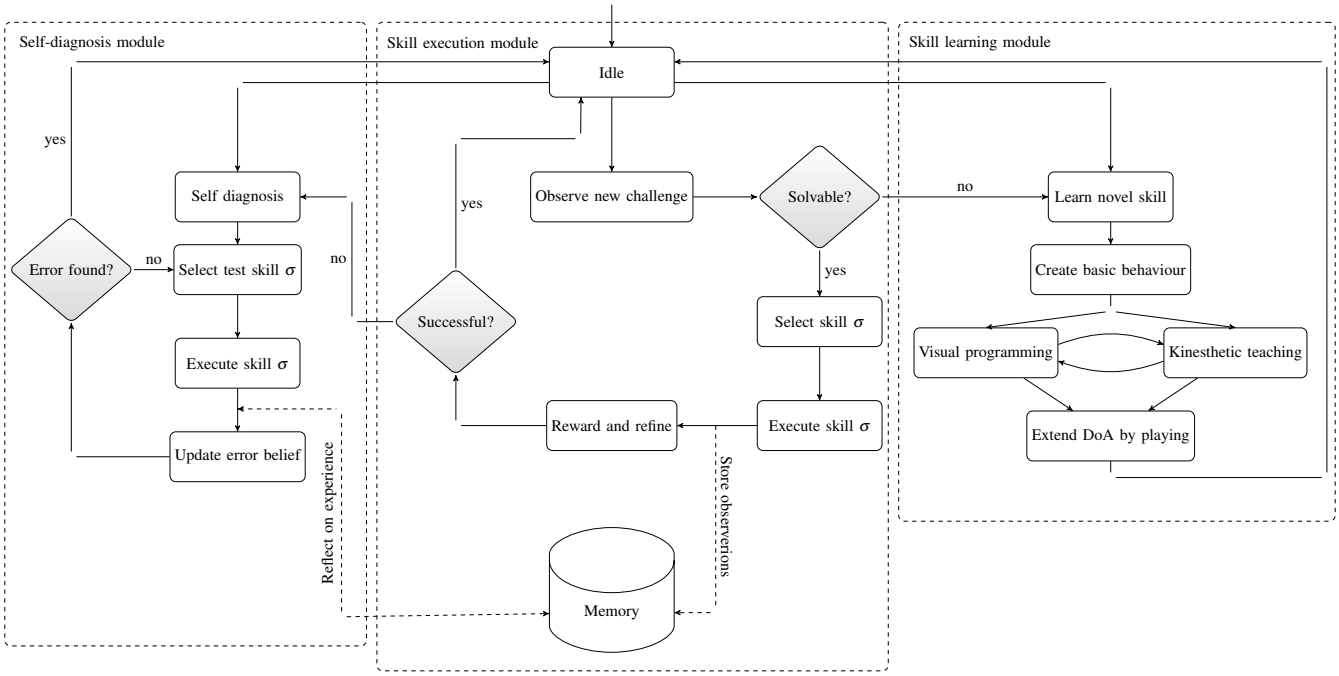


Fig. 1: System architecture illustrating the life-long learning scheme within our programming paradigm. It consists of three major parts, i.e. the *skill execution module*, the *self-diagnosis module* and the *skill learning module*. The memory is another core component of our system and stores data required by other components, e.g. sensor data or function call profiles.

the execution of a skill σ and is given by

$$\mathbf{F}_\sigma(\mathbf{s}) = \begin{pmatrix} fc_1(0) & \dots & fc_1(T) \\ fc_2(0) & \dots & fc_2(T) \\ \vdots & \ddots & \vdots \\ fc_F(0) & \dots & fc_F(T) \\ t=1 & \xrightarrow{\Delta t} & t=T \end{pmatrix} \begin{matrix} \text{function } 1 \\ \text{function } 2 \\ \\ \downarrow \\ \text{function } F \end{matrix} \tag{4}$$

The database contains information on which skill requires certain hardware configurations and both matrices are automatically stored to a database when a skill is executed.

C. System architecture

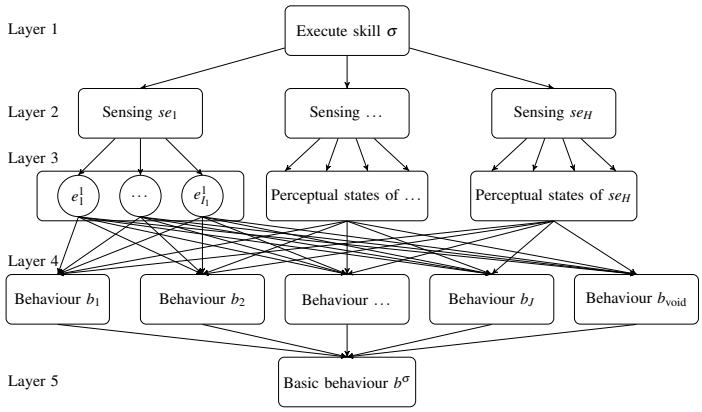
This section describes the key ingredients and their interplay in the context of the proposed programming paradigm and is independent of concrete methods. The system architecture is shown in Fig. 1. Our life-long learning / programming approach consists of three major parts, i.e. *skill execution*, *skill learning* and *self-diagnosis*.

1) Skill execution module: The skill execution module is responsible for executing well-trained skills. The robot is confronted with a task and a decision on whether or not the task can be solved with the available skill repertoire is made. If a task cannot be solved, the skill learning module is called, which involves external input from a programmer or teacher, c.f. section IV-C.2. If the task is solvable, the appropriate skill is selected and executed by performing a behaviour sequence $b_k \circ \ldots \circ b_1 \circ b^\sigma(s)$ in the current state $s \in S$ which solves the task. The execution is rewarded and the models are refined if the execution scheme supports this. During the execution of skills all experiences, i.e. sensor data and function call profiles, are stored in the memory of the robot in order to use it for learning and self-diagnosis.

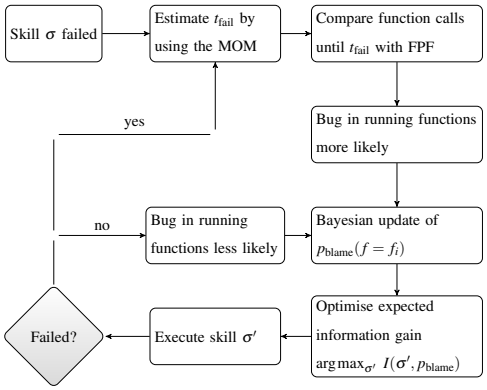
2) Skill learning module: The skill learning module can either be invoked out of idleness of the robot or if a presented challenge cannot be solved by the robot. The goal is to acquire a novel skill σ with a large DoA. In our setting, this requires two parts, i.e. teaching of the basic behaviour b^σ and identification of state dependent behaviour sequences $b_k \circ \ldots \circ b_1 \circ b^\sigma(s)$ which enlarge the DoA. In this work, basic behaviours can be implemented by visual programming (section VII-E), by kinesthetic teaching or a mixture of both. In order to enlarge the DoA of the novel skill we propose to use autonomous playing, c.f. section V.

3) Self-diagnosis module: We emphasise the need of a way to autonomously identify bugs in the software as part of the programming paradigm. The self-diagnosis module is called if the robot is idle or if typically successful skills start to fail. Due to the life-long programming approach, the robot has access to a large database of experiences, i.e. sensor data and function call profiles. Skills are executed as test cases in order to identify functions and behaviours which cause a problem that was not present before certain code changes. The system returns a distribution $p_{\text{blame}}(f = f_i)$ defining the probability of the function f_i causing the problem. In this paper we use the testing approach described in section VI.

4) Memory: The memory is a core component of our architecture following the philosophy of storing all available experiences made over the life-time of the robot. Previous work has demonstrated a way on how to store sensor data



(a) Schema of the episodic and compositional memory (ECM) for a certain skill σ. Skills are executed by performing a random walk through the ECM according to the probabilities that are assigned to the transitions. A skill execution consists of the sequence *sensing action* → *perceptual state estimation* → *environment preparation* → *basic behaviour execution*. Skills are trained by using *projective simulation*.



(b) Flow diagram of the used autonomous self-diagnosis approach. A *measurement observation model* (MOM) for sensor data and a *functional profiling fingerprint* (FPF) for function call profiles is trained for each skill from observations during successful executions.

Fig. 2: Conceptual design on the used implementations for the *skill execution module* (Fig. 2a) and the *skill learning module* (Fig. 2b). The overall architecture is shown in Fig. 1.

to the non-SQL database MongoDB based on listening on ROS topics and using this information for fault analysis [16]. We chose a different track in which hardware interfaces are provided within our software that require developers of the control classes to provide a SQL schema and *store* functions for storing a current snapshot of the sensor data. For all hardware used for the execution of a skill, the *store* function is called regularly. Further, for all framework functions the start and end time of the execution is stored in the database.

V. Skill acquisition by autonomous playing

The previous sections are agnostic of the concrete skill acquisition and self-diagnosis method. In this section we provide a brief description of the skill learning method used in our prototype. The reader may refer to previous work for more in-depth treatment [1], [2]. The idea is teach a novel skill σ by providing a basic behaviour b^σ and to extend the DoA by learning how to prepare a situation in which the basic behaviour can be applied without changes. The robot tries out different combinations of preparatory behaviours in different situations. We refer to this as autonomous playing. The playing consists of two stages - in the first stage so-called *perceptual states* $e \in E^{se}$ are trained from interacting with the object by using sensing actions $se \in SE$. Perceptual states are discrete concepts derived from continuous sensor observations and capture the task-relevant information. This might require different sensing actions for different tasks, e.g. poking on top of a box can be used to identify whether or not a box is open, whereas sliding along the surface can be used to determine the orientation of a book. Different situations are prepared either by the robot or by a human teacher and the robot creates a haptic database of sensor data observed in different perceptual states. A SVM is trained from the sensor data for perceptual state classification.

After this first playing phase a so-called *episodic and compositional memory* (ECM) of the form shown in Fig. 2a is constructed. The ECM is the basis of *projective simulation* [17] and consists of *clips* that are connected by transitions with certain transition probabilities. A skill is executed by a *random walk* through the ECM and by executing the actions along the path. The transition probabilities along a rewarded path are increased, i.e. if the success predicate of the skill σ is fulfilled. This forces the robot to learn (i) which sensing action provides the best perceptual states; (ii) how to prepare the environment correctly in a certain perceptual state. When a skill is well-trained, i.e. the average success rate during playing is high, it can be used as preparatory behaviour for other skills to train in the future. This enables to robot to create skill hierarchies and to learn how to solve increasingly complex problems. The described basic version was extended by creative planning and active learning mechanisms in followup work [1]. While playing, a simple environment model is trained which is used to suggest potentially useful preparatory behaviours to the model-free playing system. An in-depth treatment is provided in previous work [1], [2].

VI. Autonomous skill-centric testing

For *self-diagnosis module* we use a method for autonomous testing developed in previous work [3]. During a skill execution, an observation $o = (\mathbf{M}_\sigma(s), \mathbf{F}_\sigma(s))$ is made. From the matrices $\mathbf{M}_\sigma(s)$ (sensor data) and $\mathbf{F}_\sigma(s)$ (function call profiles) described in section IV-B the measurement observation model (MOM) $p_\sigma(succ \,|\, \mathbf{M}, t)$ and the functional profiling fingerprint (FPF) $p^f_\sigma(fc \,|\, \mathbf{M}, t, succ = \text{true})$ for each software function f are trained. The variables $s \in S$, $succ$, t and fc denote the environment state, the execution success, the time step and the number of running instances of the function f respectively. The MOM and the FPF are trained by an encoder / decoder neural network and by a simple multivariate Gaussian model respectively. A sketch of the testing scheme is shown in Fig. 2b. If a skill σ failed, the MOM is used to identify the time t_{fail} at which the error

occurred. All functions that were running until this point are at least suspicious proportional to the time distance to t_{fail}, and even more so if the matrix $\mathbf{F}_\sigma(\mathbf{s})$ for a certain function strongly deviates from the trained FPF. If a function is running during a successful execution, it becomes less likely that this function has a problem. From this information the likelihood function $p_\sigma(o \mid f = f_i)$ is computed, and a Bayesian belief update is performed with

$$p_{\text{blame}}(f \mid o_{1:T}, o, \sigma_{1:T}, \sigma) \propto p_\sigma(o \mid f) p_{\text{blame}}(f \mid o_{1:T}, \sigma_{1:T}) \quad (5)$$

In a further step, our system selects the next skill σ' to execute by maximising the expected information gain $\mathbf{E}[I(\sigma')]$ about which function causes problems. The resulting distribution $p_{\text{blame}}(f = f_i)$ is estimated autonomously. The developer can then either fix the bug if the knowledge is available or report it to the responsible designer of the respective components.

VII. System integration

In following section we describe practical considerations on essential key components in order to achieve two major goals: the *integration of skill acquisition and the autonomous testing* and the *design of the visual programming framework*.

A. The kukadu framework

All the described methods are implemented in *kukadu* - a framework for autonomous robotics. *kukadu* provides several modules required in robotics:

- Hardware control: Provides interfaces for hardware components like arms or depth-image cameras. Implementations for certain hardware are included.
- Path planning: Provides interfaces for path planning (joint space and Cartesian space) and kinematics. Bridges to path planning frameworks such as *MoveIt*[5] or *Komo* [18] are available.
- Control policies: Provides interfaces for parametrised control policies and standard methods such as DMPs (joint space [19] and Cartesian space [20]) are implemented. Policies can be trained by Kinesthetic teaching.
- Machine learning: Different standard machine learning techniques such as linear regression, Gaussian process regression, Support vector machines [21], unsupervised clustering are available. Certain policy reinforcement learning approaches like PI^2 [15] or PoWER [14] and gradient descent methods are available. The algorithms are connected to the control policy interfaces.
- Computer vision: Interfaces for object localisation and pose estimation are defined. Elementary object localisation, integrated with the depth-image interfaces, based on fitting boxes to segmented point clouds is provided.
- Autonomous robotics: The approaches described in sections V and VI are implemented and connected to the available hardware interfaces. The methods are based on a skill interface which provides automatic controller generation or automatic storage of sensor snapshots.

We further provide a visual programming interface for unexperienced developers and programmers without C++ knowledge. The following sections explain key concepts on an abstract level that make this possible inside *kukadu*. Detailed information can be found in the framework documentation.

B. SQL-based robot memory

Many ROS-based applications use the integrated MongoDB in order to store data. This approach avoids the need for designing an SQL schema for robotics applications. We chose a different approach by designing an extensible SQL schema, considering typical data provided by hardware that is interfaced in the framework. Certain hardware provides more information, e.g. new robotic arms might provide more or different sensor data, which is why implementations of hardware interfaces can provide their own additional schema in form of SQL files, which are installed on program startup. Similar concepts hold for skill controllers, which are installed automatically, e.g. storing the required hardware configurations. More involved skill controllers can further install custom SQL schema and store additional information.

C. Factories

In order to provide easy drag and drop functionality for end-users, it must be easy to instantiate hardware control and skill objects without having to deal with constructors. This is ensured by hardware and skill factories, which query the code on how to generate instances from a code database provided by the respective developers of the interfaces. Custom code for loading required properties from the database can also be defined. Hardware can be created by providing a concrete instance name, e.g. *left_arm*. Skills can be created by defining the skill name and the desired hardware configuration, e.g. the *simple grasping* skill in Fig. 3a requires a hardware configuration consisting of *left_arm, camera* and *left_hand*. Further, the factory makes sure that only one controller instance exists at the same time per hardware component.

D. Skill interface and data acquisition

A skill is generated by implementing an abstract skill controller class. An instance of the respective controller is created by the skill factory using the information stored in the database. If a skill is executed, pre-implemented functions query all the required hardware. A monitor regularly requests the used hardware to store snapshots of the available sensor data to the database. This is especially important for the generation of the skill-specific FPF and MOM, c.f. sections IV-B and VI. The sensor data is connected to the respective starting and end times of concrete executions and to the success information.

E. Visual robot programming

The visual programming tool enables the user to create simple robot programs based on drag and drop. We base our implementation on the *cake* framework[6], which generates C++ code from the graphic input. In principle arbitrarily

[5] http://moveit.ros.org/

[6] https://github.com/cra16/cake-core

complex programs can be generated and typical language constructs like loops or functions are available out of the box. We further added custom blocks for robotic programming, namely *skill* blocks and *hardware* blocks. The programmer can drop multiple hardware components onto the canvas and the interface suggests skills that are available for a certain hardware setting, e.g. by using the arm, the hand and the camera the *grasping* skill can be used.

We provide an initial set of different simple skills ranging from simple point to point movement (Ptp) skills to more high-level skills like pushing. Some skills require additional input which can be defined as well, e.g. the *JointPtp* skill needs the desired goal position. *kukadu* is implemented in C++ which does not come with a sophisticated reflection API such as *Java*[7]. Therefore, the supported properties of certain skills are extracted by generating an XML-based representation of all member functions of the skill implementation classes by using *Doxygen*[8].

1) Skill implementation: The developer can generate the code for a novel skill controller and test it automatically on the robot. We support an interactive programming paradigm in which programming can be done directly in a certain environment by a mix of visual programming and kinesthetic teaching. In the task of placing a book to a shelf the developer can use the *book grasping* skill to grasp the book. The *book placement* skill can be tested up until this point which leaves the robot in a state where the book is in the robot's hand. The user can continue by showing how to place the book in the shelf by kinesthetic teaching. This will work only if the book in the correct pose, c.f. section V. Instead of asking the user to consider even more cases, one can create a basic behaviour out of the visual program and call the module for autonomous playing. The correct treatment of different book orientations, i.e. the extension of the DoA of the *book placement* skill will then be learned by the robot.

If the user is happy with the result the novel skill can be installed and used for the implementation of other skills as well. This way more and more complex skills can be implemented and skill hierarchies can be created.

2) Skill testing: During training and the usage of skills a large amount of sensor data is automatically stored in the database. If code is changed this could influence the implemented skills, e.g. if the skill for moving the arm in Cartesian space starts using a different planner. Such errors can be identified by the skill testing module [3] which is integrated as a separate tool. The information on potentially problematic functions can be used by the developer to consult experts on the respective components or to change used parameters. This strongly reduces the need for the user to be familiar with certain components in detail.

VIII. Evaluation

The separate components of our approach to robot programming, namely the *autonomous playing* [1], [2] and the *skill centric testing* [3], were evaluated in their respective publications. In this paper we investigate the applicability of the complete and integrated system based on our novel robot programming paradigm. We demonstrate that a wide variety of skills can be implemented with our framework for visual robot programming. In order to evaluate the skill acquisition, we implemented basic behaviours for different skills by visual programming. The DoA is then extended by autonomous playing.

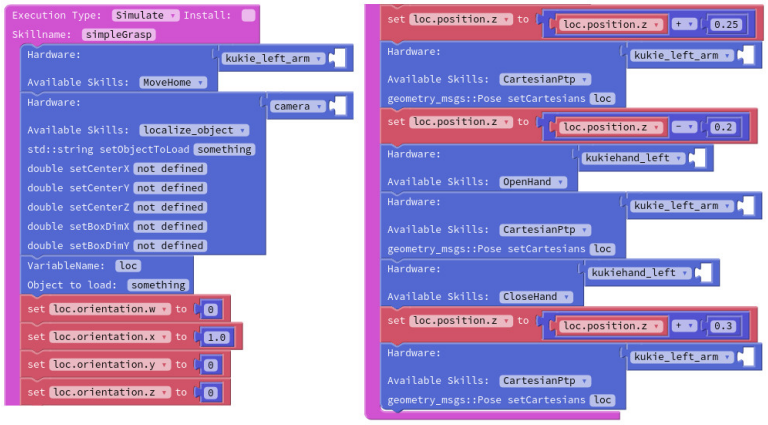
Name	Description	Type	Visual
Sliding	Slides along the object's surface	Se	Yes
Poking	Pokes on top of the object	Se	Yes
Pressing	Presses the object between the hands	Se	Yes
Joint movement	Moves joints to specified position	B	No
Cartesian movement	Moves end-effector to Cartesian position	B	No
Move home	Move to initial position	B	Yes
Close hand	Close the hand	B	Yes
Open hand	Open the hand	B	Yes
Change stiffness	Change impedance settings of the arm	B	No
Push to body	Push an object towards the body	B	Yes
Push from body	Push object away from the body	B	Yes
Push to position	Push object to a certain position	B	Yes
Push to orientation	Rotate the object to a certain orientation	B	No
Simple grasp	Place end-effector on top of an object and close the hand	B	Yes
Pick and place	Grasp an object and place it in a box	B	Yes
Press button	Presses a button at a fixed position	B	Yes
Book grasping	Grasp a book from a rigid table	Sk	Yes
Shelf placement	Place an object in a shelf	Sk	Yes
Tower disassembly	Disassembles a tower of boxes	Sk	Yes
Localise object	Searches for a defined object on the table	B	No

TABLE I: List of the implemented skills and behaviours by using the *kukadu* framework. The *Visual* column indicates whether or not the corresponding controller was implemented by visual programming. The *Type* corresponds to the controller type (*Se* $\equiv$ sensing action, *B* $\equiv$ behaviour, *Sk* $\equiv$ skill trained with autonomous playing).

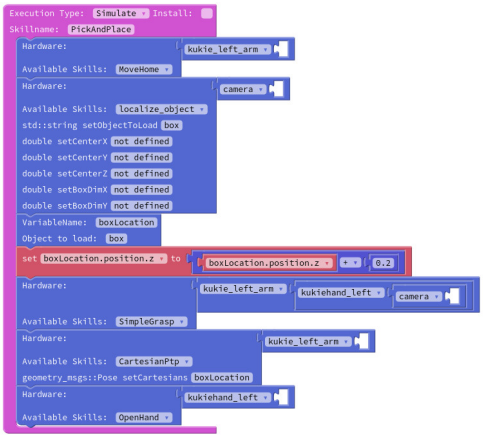
A. Robot setting

The used robot setting is shown in Fig. 4. It consists of two KUKA LWR 4+ robotic arms mounted on a metal pillar. A Schunk SDH gripper is mounted to each one of the robot's arms. A Kinect camera is mounted above the robot for object recognition and localisation. In order to develop and test the implemented skills several objects from the YCB dataset

[7] https://docs.oracle.com/javase/tutorial/reflect/
[8] http://www.stack.nl/~dimitri/doxygen/



(a) Visual program of a *simple grasping* skill. The robot moves to its' home position, detects some object on the table, moves the end-effector to that position and closes the hand.



(b) Visual program of a *pick and place* skill which reuses the simple grasping skill in Fig. 3a.

Fig. 3: Exemplary visual programs of a *simple grasping* skill and a *pick and place* skill. The creation of skill hierarchies, e.g. the *pick and place* skill uses the *grasping* skill, is demonstrated.

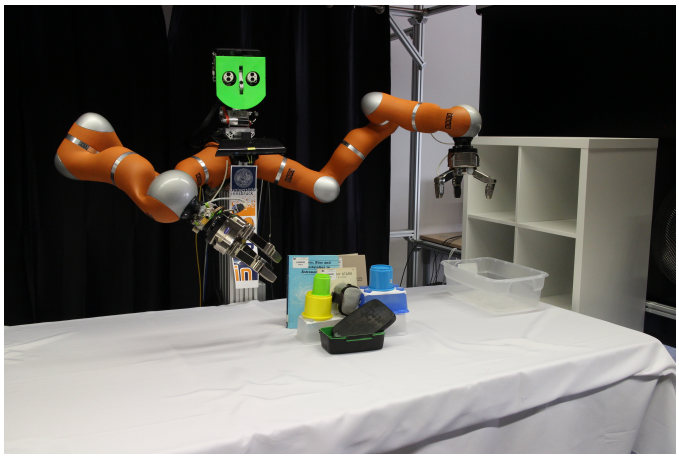


Fig. 4: The robot setting (2 KUKA LWR 4+ with attached Schunk SDH grippers) and the used objects.

were used [22]. We further used other objects such as books of different sizes and Ikea furniture and boxes.

B. Implemented skills

We demonstrate the generality of our approach by implementing a wide range of different skills and behaviours by visual programming and autonomous playing. A list of these skills including a short description can be found in Table I.

1) Conventional behaviour programming: Fig. 3a shows a *simple grasping* skill in which the object is localised and the end-effector is moved on top of it. The hand is closed and the object is lifted. The skill can directly be tested and installed if the user is satisfied with the result. It is immediately available for further use, e.g. with the *pick and place* skill shown in Fig. 3b. This way skill hierarchies can be created.

We further implemented a *push to position* behaviour which demonstrates how more complex control structures such as loops can be used. The end-effector follows a Cartesian path from the object's current position to the desired position. The path is planned in Cartesian space in a loop with intermediate milestone positions.

2) Implementation of skills by autonomous playing: Up until now none of the described behaviours required autonomous playing. We implemented a *book grasping* skill and a *tower disassembly* skill. The basic behaviour for *book grasping* is programmed according to the description in section V, c.f. video[9]. The book is localised by using the *localise object* behaviour and is pushed to a position in front of the robot with the *push to position* behaviour. The basic behaviour can be executed up until this point and the user can use kinesthetic teaching in order to push the book towards the robot until it is squeezed between both hands. The system automatically learns a DMP out of the demonstrated trajectory and makes it available for further use during visual programming. The user can then continue to implement the rest of the basic behaviour.

This approach only works if the book is rotated correctly such that it can be lifted at the spine. The robot learned how to deal with different rotations by autonomous playing. It automatically identified that the *sliding* behaviour is best suited to estimate the book's orientation and that pushing the book to the correct rotation yields success.

The skill of *disassembling a tower of boxes* skill shows that the basic behaviour can even be left empty, c.f. video[10]. This way the robot learns how to coordinate already existing behaviours and figures out that the tower can be disassembled by performing h consecutive *pick and place* actions, where h is the height of the tower. The height can be estimated by *poking* on top of the tower.

Some skills were not programmable with visual programming due to the restrictions in order to provide a simple

[9] https://iis.uibk.ac.at/public/shangl/iros2016/iros.mpg

[10] https://iis.uibk.ac.at/public/shangl/iros2016/iros.mpg

programming framework. This can be the case if complex sensor data processing or hardware functions are required. An example is the *rotation by pushing* skill which requires fine-grained direct control of the shape of the trajectory. Another example is the *change stiffness* skill which changes the stiffness settings of the robot's impedance mode. In such cases the skills can be implemented by experts and can be made available to unexperienced programmers for usage.

C. Testing

The testing functionality can be called as a separate tool in case the robot either keeps failing on a skill that was well-trained before or if the user triggers it manually. The testing framework is especially helpful in case the user changes the implementation of a skill or behaviour, e.g. if properties of used behaviours are changed or if the structure of the implemented behaviour is changed completely. An example could be that the used planner for executing a Cartesian plan was switched. Suggestions of functions that may have caused the problem are then made. These can be fixed by the user or forwarded to experts in the respective fields. In prior work we showed that such errors can be found autonomously [3].

IX. Conclusion and Outlook

In this work we presented a novel paradigm for robot programming which adopts technology developed for developmental robotics in order to ease the task for robot programmers. The programmer only has to implement a basic behaviour that solves a task for only one situation by interactive programming with visual programming and kinesthetic teaching. The robot then uses this program as a starting point in order to extend the domain of applicability.

Further, the robot gathers a lot of sensor data during the execution of skills which can then be used to identify bugs in the software if a skill stops working.

We evaluated our approach by implementing several behaviours and skills with our new tool for visual programming and extended the DoA by autonomous playing.

There are many ways along which our approach can be extended. If one looks at the generated programs, many of the commands can be represented as a simple sentence with a *subject*, e.g. camera, a *predicate*, e.g. localise, and an *object*, e.g. book. This could be exploited to implement a verbal interface which could be used in combination with visual programming. Further, we plan to extend our developmental approaches in order to enable the robot to learn the extension of the DoA more efficiently, e.g. by transferring knowledge between skills. Another interesting direction is to transfer skills between different robots, which in general would enable robot programmers to implement skills which can easily be shared with other robot owners.

References

[1] S. Hangl, V. Dunjko, H. J. Briegel, and J. Piater, "Skill learning by autonomous robotic playing using active learning and creativity," e-Print arXiv:1706.08560, 2017.

[2] S. Hangl, E. Ugur, S. Szedmak, and J. Piater, "Robotic playing for hierarchical complex skill learning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2016.

[3] S. Hangl, S. Stabinger, and J. Piater, "Autonomous Skill-centric Testing using Deep Learning," in *IEEE/RSJ International Conference on Intelligent Robots and Systems*, 2017.

[4] H. Lieberman, F. Paternò, M. Klann, and V. Wulf, "End-user development: An emerging paradigm," in *End user development*. Springer, 2006, pp. 1–8.

[5] A. J. Ko, B. A. Myers, and H. H. Aung, "Six learning barriers in end-user programming systems," in *Visual Languages and Human Centric Computing, 2004 IEEE Symposium on*. IEEE, 2004, pp. 199–206.

[6] M. Resnick, J. Maloney, A. Monroy-Hernández, N. Rusk, E. Eastmond, K. Brennan, A. Millner, E. Rosenbaum, J. Silver, B. Silverman, *et al.*, "Scratch: programming for all," *Communications of the ACM*, vol. 52, no. 11, pp. 60–67, 2009.

[7] S. H. Kim and J. W. Jeon, "Programming lego mindstorms nxt with visual programming," in *Control, Automation and Systems, 2007. ICCAS'07. International Conference on*. IEEE, 2007, pp. 2468–2472.

[8] J. Jackson, "Microsoft robotics studio: A technical introduction," *IEEE Robotics & Automation Magazine*, vol. 14, no. 4, 2007.

[9] H. Nguyen, M. Ciocarlie, K. Hsiao, and C. C. Kemp, "Ros commander (rosco): Behavior creation for home robots," in *Robotics and Automation (ICRA), 2013 IEEE International Conference on*. IEEE, 2013, pp. 467–474.

[10] F. Riedo, M. Chevalier, S. Magnenat, and F. Mondada, "Thymio ii, a robot that grows wiser with children," in *Advanced Robotics and its Social Impacts (ARSO), 2013 IEEE Workshop on*. IEEE, 2013, pp. 187–193.

[11] S. Alexandrova, Z. Tatlock, and M. Cakmak, "Roboflow: A flow-based visual programming language for mobile manipulation tasks," in *2015 IEEE International Conference on Robotics and Automation (ICRA)*, May 2015, pp. 5537–5544.

[12] S. Alexandrova, M. Cakmak, K. Hsiao, and L. Takayama, "Robot programming by demonstration with interactive action visualizations." in *Robotics: science and systems*, 2014.

[13] S. Hangl, E. Ugur, S. Szedmak, A. Ude, and J. Piater, "Reactive, Task-specific Object Manipulation by Metric Reinforcement Learning," in *17th International Conference on Advanced Robotics*, 7 2015.

[14] J. Kober and J. R. Peters, "Policy search for motor primitives in robotics," in *Advances in Neural Information Processing Systems 21*, D. Koller, D. Schuurmans, Y. Bengio, and L. Bottou, Eds. Curran Associates, Inc., 2009, pp. 849–856.

[15] E. Theodorou, J. Buchli, and S. Schaal, "A generalized path integral control approach to reinforcement learning," *Journal of Machine Learning Research*, vol. 11, no. Nov, pp. 3137–3181, 2010.

[16] T. Niemueller, G. Lakemeyer, and S. S. Srinivasa, "A generic robot database and its application in fault analysis and performance evaluation," in *2012 IEEE/RSJ International Conference on Intelligent Robots and Systems*, Oct 2012, pp. 364–369.

[17] H. J. Briegel and G. De las Cuevas, "Projective simulation for artificial intelligence," *Scientific Reports*, vol. 2, pp. 400 EP –, May 2012, article.

[18] M. Toussaint, "KOMO: Newton methods for k-order markov constrained motion problems," e-Print arXiv:1407.0414, 2014.

[19] S. Schaal, "Dynamic movement primitives-a framework for motor control in humans and humanoid robotics," in *Adaptive motion of animals and machines*. Springer, 2006, pp. 261–280.

[20] A. Ude, B. Nemec, T. Petrić, and J. Morimoto, "Orientation in cartesian space dynamic movement primitives," in *Robotics and Automation (ICRA), 2014 IEEE International Conference on*. IEEE, 2014, pp. 2997–3004.

[21] C.-C. Chang and C.-J. Lin, "LIBSVM: A library for support vector machines," *ACM Transactions on Intelligent Systems and Technology*, vol. 2, pp. 27:1–27:27, 2011.

[22] B. Calli, A. Singh, A. Walsman, S. Srinivasa, P. Abbeel, and A. M. Dollar, "The ycb object and model set: Towards common benchmarks for manipulation research," in *Advanced Robotics (ICAR), 2015 International Conference on*. IEEE, 2015, pp. 510–517.