robotics

MDPI

Article

Leveraging Graphical User Interface Automation for Generic Robot Programming

Tudor B. Ionescu

Human-Machine Interaction Group, Vienna University of Technology, 1040 Vienna, Austria; tudor.ionescu@tuwien.ac.at; Tel.: +43-(1)-58801-33001

Abstract: A novel approach to generic (or generalized) robot programming and a novel simplified, block-based programming environment, called "Assembly", are introduced. The approach leverages the newest graphical user interface automation tools and techniques to generate programs in various proprietary robot programming environments by emulating user interactions in those environments. The "Assembly" tool is used to generate robot-independent intermediary program models, which are translated into robot-specific programs using a graphical user interface automation toolchain. The generalizability of the approach to list, tree, and block-based programming is assessed using three different robot programming environments, two of which are proprietary. The results of this evaluation suggest that the proposed approach is feasible for an entire range of programming models and thus enables the generation of programs in various proprietary robot programming environments. In educational settings, the automated generation of programs fosters learning different robot programming models by example. For experts, the proposed approach provides a means for generating program (or task) templates, which can be adjusted to the needs of the application at hand on the shop floor.

Keywords: generic robot programming; GUI automation; robotic process automation; graphical programming; block-based programming

check for updates

Citation: Ionescu, T.B. Leveraging Graphical User Interface Automation for Generic Robot Programming. *Robotics* **2021**, *10*, 3. https://dx.doi.org/10.3390/robotics10010003

Received: 26 November 2020
Accepted: 23 December 2020
Published: 25 December 2020

1. Introduction

Recent advances in safe, collaborative robotics technology and the increasing availability of open source robot hardware facilitated the emergence of numerous new vendors entering the market with relatively inexpensive collaborative industrial robots. While this development is welcome in a field that has been dominated by a few global players for decades, the diversity of the often proprietary, vendor-provided robot programming languages, models, and environments that are provided with these new robots is also increasing rapidly. Although most vendors appear to pursue a common goal in creating more intuitive means for programming robots using web-based programming environments, tablet-like devices as teach pendants, and concepts from the domain of smart phone user experience and design, the resulting programming languages, models, and environments are rarely compatible with one another. This poses challenges to application integrators, who strive to innovate their robot fleet while reusing existing knowledge, expertise, and code available in their organizations. In non-industrial applications, the heterogeneity of proprietary robot systems increases the need for specialized expertise and thus hinders the democratization of the technology—that is, making it more accessible to diverse non-expert users by simplifying (without homogenizing) programming and lowering overall costs of operation. In this context, generic robot programming can help to cope with the heterogeneity of robot systems in application scenarios, which go beyond those envisioned by robot vendors.

Generic (or generalized) robot programming (GRP) refers to a programming system's support for writing a robot program in one programming language or model for which there are means for converting the program into vendor-specific program instructions or commands [1–3]. By "program instructions or commands" it is meant both source code, which needs to be compiled or interpreted before runtime, and instructions or messages, which are interpreted by a target robot system or middleware at runtime. When online programming is used, the instructions are sent to the robot immediately; whereas in offline programming, an entire program is generated for a specific target system first, and then downloaded (or deployed) to the robot [1,2]. The concept of GRP stands at the basis of the so-called skill-oriented robot programming approach [2], which proposes a layered architecture, in which "specific skills" (i.e., task-level automated handling functions, such as pick and place) are created on top of so-called "generic skills" (i.e., generic robot behaviors like motions), which interface the robot-independent specific skills with robot-dependent machine controllers and sensor drivers [2]. GRP often uses code generation for interfacing different robotic systems with robot-independent path planning, computational geometry, and optimization algorithms into a layered architecture providing a "generation layer" with open interfaces [3]. Modern frameworks, such as RAZER [4], take the GRP approach one step further by integrating skill-based programming and parameter interface definitions into web-based human-machine interfaces (HMIs), which can be used to program robots on the shop floor. In all these approaches, the GRP concept is used to enable interfacing higher level robot software components (i.e., skills), which can be created in various programming languages, with different robots. Various commercial and open source GRP tools (e.g., the Robot Operating System (ROS) [5], RoboDK [6], Siemens' Simatic Robot Integration [7], "Drag&Bot" [8], RAZER [4], etc.), which are commonly used in the industrial domain, require the development of robot drivers or code generators for each of the supported robots. Once such drivers or generators exist, users are able to write generic robot programs in one language and/or programming model (e.g., C++ or Python in the case of ROS, or a graphical language or Python in RoboDK), whereas the system automatically converts generic programs to robot-specific programs or instructions.

There are several problems with these approaches that the current paper seeks to address. The development of (reliable) robot-specific drivers and code generators is costly and often left to the open source community (e.g., in the case or ROS). Developing a robot driver or code generator for one of the existing GRP environment requires expert knowledge "beyond just software, including hardware, mechanics, and physics" [9] (p. 600), and takes on average up to 25% of the entire development time [10], especially in the case of online programming, when (near) real-time communication with the robot is required (e.g., as it is the case with ROS). As García et al. [9] note, while the members of the robot software developer community tends to use the same frameworks, languages, and paradigms (i.e., ROS, C++, Python, object-oriented and component-based programming), there appears to be little awareness within this community concerning the importance of software engineering best practices for improving the quality and maintainability of software in order to make it reusable and reliable. In addition, scarce or lacking documentation of published robot software components [11,12] hinders the synthesis of a set of common principles for developing drivers and code generators.

Due to the inherent difficulties of covering a multitude of robots using a single programming model, graphical GRP environments use a reduced subset of control structures. For example, RoboDK does not support *if-then-else* and other basic conditional statements. To use more advanced programming structures, users need to switch to textual programming. In this context, with new and inexpensive (collaborative) robots entering the market at an unprecedented pace, there is a need for new GRP approaches, which (1) allow the inexpensive, flexible integration of new robots that come with their own programing models, (2) lower the cost of developing new and reliable robot drivers or code generators, (3) simplify robot programming while providing a complete set of programming structures to maximize flexibility and thus to foster the wide adoption of robots in and beyond industrial contexts, and (4) succeed in generating programs for closed robot programming environments, which use binary or obfuscated proprietary program formats.

Against this background, this paper introduces a novel approach to GRP, which leverages graphical user interface (GUI) automation tools and techniques. Modern GUI automation tools use computer vision and machine learning algorithms to process visual elements in GUIs in near real time and to emulate user interactions in reaction to those events. The proposed approach uses GUI automation to couple third-party programming tools with proprietary (graphical) robot programming environments by enabling program generation in those target environments while bypassing vendor-provided application programming interfaces (APIs). In this sense, the contribution of the paper is twofold. First, it introduces a novel web-based graphical robot programming environment, called *Assembly*, which is used to generate an intermediary-level program model that conforms to a generic robot programming interface (GRPI). Second, it illustrates both technically and methodologically how GUI automation can be leveraged to generate robot programs in various proprietary robot programming environments. As part of the approach, the programming process is divided into a generic, robot-independent step, which can be carried out anywhere using the web-based *Assembly* tool. In a second step, a robot-independent GUI automation model, corresponding to the program, is generated using the *Assembly* tool. In a third step, the user can connect a computer running a GUI automation tool, such as SikuliX [13] or PyAutoGUI [14], to a target robot programming environment (e.g., an HMI hosted on a teach pendant) to generate a program in that environment. This saves time and allows for maintaining a unified code base, from which programs for proprietary target environments can easily be generated.

The design rationale behind this approach is to democratize collaborative robot programming in makerspaces and other non-industrial contexts by facilitating an easy entry into robotics using a simplified web-based, graphical programming environment and a solution to the GRP problem based on an easily understandable and applicable technology, such as GUI automation. The proposed approach thus aims to provide a viable, more accessible alternative to entrenched robotics “monocultures”, such as the expert-driven ROS ecosystem [11], and commercial ecosystems controlled by robot manufacturers, which have dominated the robot software landscape for decades. Similarly, *Assembly* aims to provide a more straightforward alternative to Blockly and Scratch—the current de facto standards for simplified programming in educational contexts. As García et al. [9] note, “[r]obots that support humans by performing useful tasks . . . are booming worldwide” (p. 593), and this is not likely to produce less diversity concerning robot users and usage scenarios. In this sense, the hopes and expectations concerning *Assembly* are not primarily to be (re)used by many but to inspire and encourage robot enthusiasts to create more diverse, accessible, inclusive, and user-friendly “do-it-yourself” robot programming tools and to share them online—ideally as web applications. The key ingredient in this configuration is, arguably, represented by a generic robot programming interface, which builds on the “convention over configuration” principle [15] to enable the loose coupling between various third-party programming models and proprietary robot programming environments through GUI automation and (possibly) other unconventional means.

The approach is evaluated based on a real application focused on generating programs for a UR5 robot (i.e., a Universal Robots’ model 5 robot) installed in an Austrian makerspace (i.e., a shared machine shop equipped with digital manufacturing technologies that are leasable by members on premise) [16]. Over the past two decades, makerspaces emerged as alternative, non-industrial locales in which advanced manufacturing technologies (such as 3D printers, laser cutters, CNC machines, etc.) can be leased on-premise by interested laypersons, who do not have to be affiliated with a specific institution. Recently, collaborative robots also started to figure in makerspaces. Here, the challenge is twofold. First, existing industrial safety norms and standards cannot be applied to the makerspace context because member applications are not known in advance [17,18]. Second, current robot programming environments are still designed with experts in mind and/or require direct access to the robot. This makes it difficult for interested laypersons to learn and practice robot programming remotely (e.g., at home) before getting access to a real robot. The approach to robot programming described in this paper specifically addresses the latter issue.

The generalizability of the approach to list, tree, and block-based programming was assessed using three different robot programming environments, two of which are proprietary. In brief, the results of this evaluation suggest that the proposed approach to generic robot programming by emulating user interactions in robot GUIs and HMIs is feasible for an entire range of graphical (block, tree, or list-based) target robot programming environments. For example, it can be used to generate programs in proprietary graphical robot programming environments, which do not allow for importing programs created using non-proprietary tools. In educational settings, the automated generation of programs in a target graphical environment also fosters learning how to program in that environment. The proposed approach can also be used to generate program or task templates, which only require the adjustment of locations and waypoints in the robot's native graphical programming environment.

2. Related Work

2.1. Graphical Robot Programming

Proprietary robot programming environments, such as Universal Robots' (UR) Polyscope [19] and RoboDK® [6], provide rich graphical interfaces for creating procedural programs organized in a tree-like structure. These programming environments require a certain degree of computational thinking [20] on the part of robot operators. Other approaches to programming cobots recognize the need for operators to also (re)program robots on the shop floor to some extent [4,8,21–24] in a way reminiscent of the *DevOps* (development and operations) practices from software engineering. As a result, an increasing number of these environments strive to be more intuitive. For example, the web-based Franka Emika Desk [21] tool follows a simple task-oriented, multi-modal programming approach. In Desk, programs can be created by dragging and dropping robot behaviors or skills, called apps, in a sequence. This process is assisted by haptic guidance of the robotic arm and control of the end effector (e.g., gripper). Other skill-oriented programming models, such as RAZER [4] and different research tools (e.g., [25,26]), focus on the provision of robot skills by experts. Within this model, which is similar to that of Franka Emika Desk, robot skills are implemented as plugin components, which embody complex automated handling functions inspired by norms such as VDI 2680 [26]. Aligning skills with norm-based automated handling functions allegedly helps to make robot programming more intuitive for manufacturing professionals [27].

Block-based programming (i.e., a programming model in which statements are represented as configurable graphical blocks linked to each other) is becoming increasingly popular for educational [28] and industrial robots [29,30]. The widely used Google Blockly [31] tool has been shown to provide an easier entry for novice programmers than traditional programming environments, such as UR's Polyscope [19] and has been integrated into ABB's RobotStudio platform [32]. By offering more flexibility and a complete palette of control structures and statements, block-based programming presents some advantages over similar app-oriented approaches, such as Franka Emika Desk, which are often limited in terms of the program structures that can be used [33]. The VDI 2680 automated function symbols have also been used to decorate and guide the design of function blocks in an experimental constraint-based programming environment [26]. The way in which blocks are connected and parameterized differs from one block-based model to another. Whereas in Blockly, each block has a correspondent in textual programming, in Choregraph [34]—the programming tool provided with Pepper and other humanoid robots—blocks are connected with each other using lines and ports in a way that is reminiscent of function block programming [35]. While line-based connections between (function) blocks facilitate the representation of parallel computation and make data flow explicit, some researchers note that Choregraph and other similar visual programming models result in "spaghetti code", which is difficult to decipher [36].

List-based programming is a simple "no-code" programming model used by some proprietary (generic) programming environments (e.g., [8,37]) as an alternative to text-based programming. List-based programming is similar to block-based programming, with the difference that blocks are represented as indented, fixed-size rows in a list. In addition, each statement (i.e., row) is usually displayed including arguments and parameter values and provides icons for reordering and deletion.

A comprehensive review of visual programming environments for end-user robot programming is provided in [36].

2.2. Generic Robot Programming

Generic robot programming environments build upon a unified programming model to support the programming and control of multiple robots from different vendors. There appear to be two main approaches employed by such environments—the robotic middleware approach and the code generation approach. A good example for the former is the Robot Operating System (ROS) [5], which supports a high number of robot systems. In ROS, robots and other systems (e.g., cameras, end effectors, etc.) are wrapped as software nodes (i.e., stand-alone components, which run in their own thread or process), which communicate with each other using the publish-subscribe pattern [38]. Programming in ROS is usually done in C++ or Python, but support for other languages also exists. ROS also supports code generation from a graphical state-machine language and system called SMACH [39]. While ROS is a de facto standard in the domain of industrial robotics, it entails several drawbacks. For example, ROS programs do not leverage the features that are already implemented in the vendor-provided end-user programming environments. ROS uses an API (application programmable interface) developed for autonomous robot functions (e.g., algorithmic path planning with obstacle avoidance and connectivity to other robots and machines) rather than for human–robot interaction. ROS also requires advanced programming skills in C++ and/or Python, which some end users (e.g., industrial engineers, assembly workers, etc.) do not typically have, and does not offer any domain-specific safety and security features, which are typically offered by proprietary programming environments. Some authors note that programs written for robotic middleware, such as ROS, are difficult to reuse due to a lack of best software engineering practices concerning the provision of ready-to-use (containerized) components in addition to source code [40]. These drawbacks suggest that, while ROS is perhaps the most advanced industrial robot programming environment, having near real-time capabilities, it also induces development costs that can render the wider adoption of robotics technology, notably that of collaborative robots, financially infeasible. In practical application scenarios, which nowadays also include non-industrial contexts, such as makerspaces [17], pragmatic solutions are needed to facilitate the development of user-friendly, non-proprietary robot programming tools [33].

Tools such as RoboDK® [6], ArtiMinds [41], and "Drag&Bot®" [8] leverage code generation techniques to implement GRP for various industrial robots. These desktop (i.e., RoboDK and ArtiMinds) or web-based (i.e., Drag&Bot) tools can be used to program a wide range of robots using a common proprietary program model and robot-specific simulators. The programs thus created can be translated into robot-specific (proprietary) programming languages and downloaded to robots of different make and model. RoboDK, for example, uses a tree-based programing model, which is similar to that of Universal Robots' Polyscope [19]. One disadvantage of these tools is that they provide a limited set of control structures (e.g., RoboDK lacks basic conditional statements, such as *if-then-else*) and limited interoperability with other programming environments (i.e., Drag&Bot cannot import or export programs in other formats). In addition, with the notable exception of ROS, generic robot programming environments are commercial, proprietary, and relatively expensive.

2.3. GUI Automation

Originally, GUI automation emerged as a technique in testing graphical user interfaces. The approach has been used at least since the early 1990's and has undergone major developments ever since. Whereas in the early phase, GUI automation tools could only manipulate the mouse and keyword of a computer in the absence of a real user by following a script indicating the exact coordinates of different graphical elements on screen (e.g., buttons, text boxes, menu items), newer tools use image recognition to infer the coordinates of the graphical elements of interest automatically. This opens up an entire range of new application possibilities, which include interfacing different systems via the GUI rather than the API of a system being controlled. Put simply, GUI automation can turn a system's GUI into an API in a non-intrusive way, thus empowering users to apply software-based systems in ways that have not been foreseen by their original creators. For example, GUI elements can be wrapped as objects [42,43] and used in other programs and models for purposes that may well go beyond testing.

Modern GUI automation tools leverage computer vision and machine learning algorithms to process visual elements in GUIs in near real time and to emulate the user interactions required to react to those events. In the past decade, GUI automation has proven effective in domains such as business process automation and public administration [44], where it is used to automate workflows in enterprise software for business operations and customer relations management as well as in mundane office software. Since both of these software categories are known to induce vendor lock-in, GUI automation—which in these domains is better known as robotic process automation (RPA) [45]—helps to avoid outsourcing repetitive task with low added value by using so-called software robots to automate them [45]. Currently, there are few known applications of GUI automation in the industrial automation domain. Such techniques have been used, for example, to abstract robotic control and perception functions and to integrate them in a GUI automation scripting language to the end of simplifying robot programming by performing visual pattern recognition of objects in a ROS-based simulation [46]. GUI automation has also been used to control the user interface of a welding robot for purposes of interoperability with other systems and testing [47]. Other applications include testing NASA's Spaceport command and control system [48], automatic configuration of automation systems [49] as well as anomaly [50] and intrusion [51,52] detection in industrial control systems. In previous work [18], we used GUI automation to enhance the safety features of robot HMIs and to facilitate interoperability with other systems, such as 3D cameras. In the educational domain, GUI automation was used to enable voice programming in Blockly [53]—an approach we also evaluated for the case of industrial robots [18].

In addition to the applications mentioned above, in the industrial robotics domain, the problems which call for a solution based on RPA are mainly due to the heterogeneity of proprietary automation software [9], which cannot be easily configured and modified by users. Hence, users are dependent on vendor-provided software updates and plugins. As a result, some companies choose to unify their robot fleets so as to use a single vendor of trust. This, however, is a premise for vendor lock-in. In this context, GUI automation offers a means to program and configure robots at a meta level, which allows for diverse robotic systems to coexist on the shop floor.

3. Solution Approach

Figure 1 illustrates the proposed approach to generic robot programming by emulating user interactions in different robot-specific graphical programming environments using GUI automation (or RPA) tools and techniques. A user first creates a source program in a non-proprietary GRP environment, such as *Assembly*, then exports a so-called RPA model (i.e., a program in the language used by the RPA tool), which can be used within an RPA environment to generate a target program in a proprietary robot programming environment (e.g., the robot's HMI provided with a teach pendant). The user can check, adjust (e.g., robot poses and waypoints), test, and use the resulting program in the target robot's HMI. As best practice, the adjustments made to the target program should also be replicated in the source program to maintain consistency.

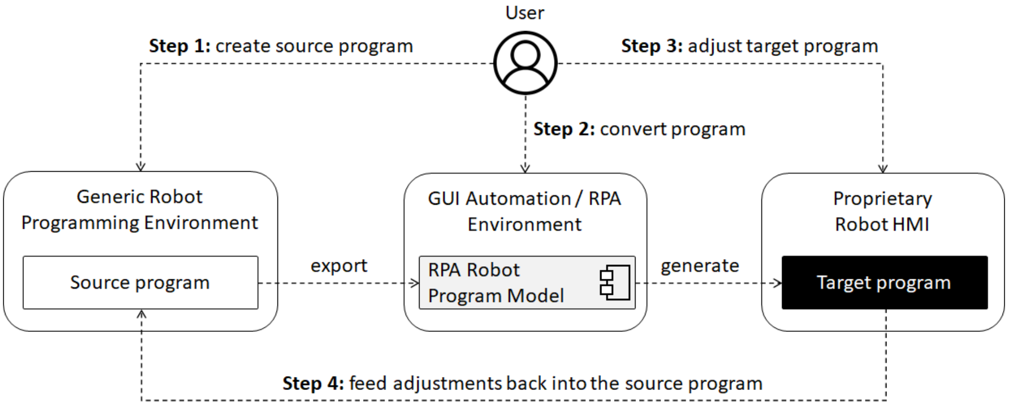

User
Step 1: create source program
Step 3: adjust target program
Step 2: convert program
Generic Robot Programming Environment
Source program
export
GUI Automation / RPA Environment
RPA Robot Program Model
generate
Proprietary Robot HMI
Target program
Step 4: feed adjustments back into the source program

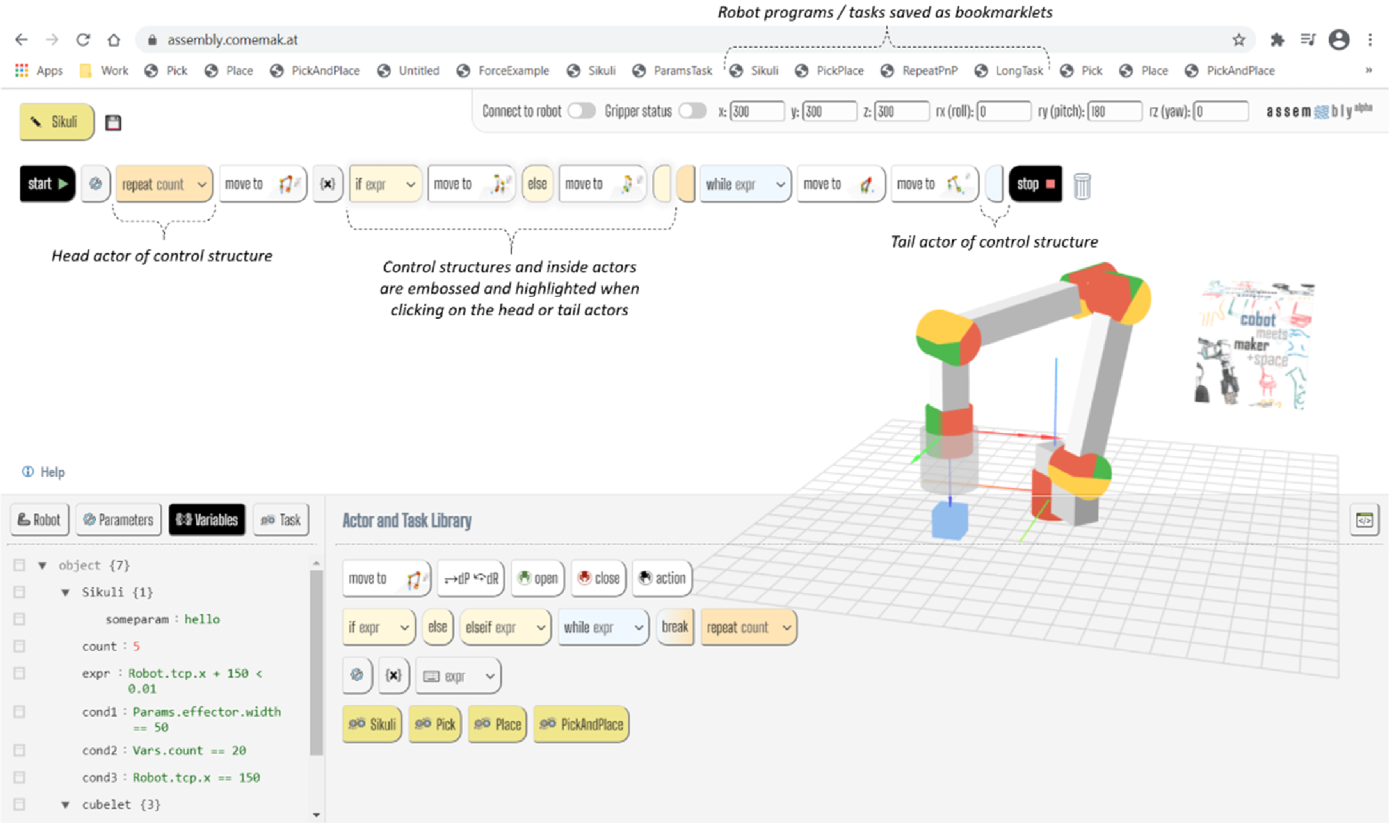
Figure 1. Procedure for generating a robot program in a target graphical programming environment.

Although parts of this process can be automated, in this paper, the detailed description of each process step will assume that access to the robot's HMI is available on the computer hosting the GUI automation tool such that a user can manually trigger the generation of the program in the target environment. In the case of UR's Polyscope [9]—the robot HMI which will be used as the main example in this paper—the URSim simulator, which is provided as a virtual machine cost free on UR's website [54], can be used as a proof of concept. To access the HMI of a real UR robot remotely, RealVNC [55] can be used to mirror UR's HMI on another computer [51]. This allows for the proposed approach to work over a direct remote connection to the target robot's HMI hosted on a teach pendant. In the following, each of these steps is described in more detail.

3.1. Assembly—A Generic Web-Based Graphical Environment for Simplified Robot Programming

Assembly is an open source simplified web-based robot programming environment, which belongs to the category of visual robot programming environments for end user development [36]. The tool builds on a new block-based programming model that is similar to—but strives to be more straightforward than Blockly (see Figure 2). *Assembly* provides an "Actor and task library", from which items (i.e., actors or tasks) can be dragged and dropped to form a sequential workflow. Actors implement basic robot behaviors, such as moving in a certain way or actuating an end effector; or program control structures, such as conditionals and loops. Actors can be added to the library by developers. A task represents a parameterizable workflow composed of actors and other tasks. Tasks can be stored as bookmarklets in the browser's bookmarks bar. A bookmarklet is a HTML link containing JavaScript commands that can be saved in the browser's "favorites" bar, which—when clicked—can enhance the functionality of the currently displayed web page [33]. In the case of *Assembly*, tasks are bookmarklets that generate a workflow corresponding to a program that has previously been saved by the user. Clicking on a task bookmarklet will also add that task to the library so that it can be used in other tasks.

The design rationale of *Assembly* is for programs to be written and read left to right, like text in most Indo-European languages. The tool can be localized to other languages and cultures by changing the progression direction of programs (e.g., from right to left or top to bottom). Unlike in the case of textual programming languages or Blockly, there is no explicit indentation, only a linear enchaining of actor instances, which form a workflow. Instead, as shown in Figure 2, rounding and highlighting effects are used to indicate how an actor relates to other actors placed before and/or after itself. A workflow may be conceived of as a phrase of varying length and complexity, which—when properly formulated—achieves the desired effect. Consequently, each actor instance may be regarded as a short sentence in that phrase. At the same time, actors encompass the semantics necessary to generate code in textual programming languages.



Robot programs / tasks saved as bookmarklets

x: 300 y: 300 z: 300 rx (roll): 0 ry (pitch): 180 rz (yaw): 0

Head actor of control structure

Control structures and inside actors are embossed and highlighted when clicking on the head or tail actors

Tail actor of control structure

```
▼ object {7}
  ▼ Sikuli {1}
      someparam : hello
    count : 5
    expr : Robot.tcp.x + 150 <
           0.01
    cond1 : Params.effector.width
            == 50
    cond2 : Vars.count == 20
    cond3 : Robot.tcp.x == 150
  ▼ cubelet {3}
```

Figure 2. *Assembly* screenshot showing the workflow of a generic robot program. The tool is available online at: https://assembly.comemak.at. See the "Supplementary Materials" section for source code availability.

Technically, a workflow is implemented as a sortable HTML list of items representing actors and tasks, the order of which can be modified by the user. To implement such a list, the "jQuery Sortable" library was used [56]. *Assembly* integrates a simple open source generic 6-axis robot simulator and inverse kinematics library [57], which can be used to define motion waypoints and to visualize robot moves. The simulator can be controlled either by dragging the end effector or by setting the coordinates of a desired waypoint using the provided controls for Cartesian coordinates and Euler angles. To memorize a certain pose as a waypoint, the user must drag the "move to" actor to the workflow after having reached the desire position using the simulator controls. Additional program logic can be implemented by dragging and dropping other actors into the workflow.

Actors implementing control structures are composed of color-coded "head" (e.g., structures such as *if*, *while*, *else*, *repeat*, etc.) and "tail" (e.g., a closing accolade "*}*") actors (or elements) corresponding to an inline instruction, such as:

1. *if(condition){* + actors/tasks + *} else {* + actors/tasks + *}*; or
2. *while(condition){* + actors/tasks + *}*

Similar to Blockly, *Assembly* helps novice programmers to avoid syntax errors by providing fixed structures that can be arranged in a sequence, whereby some program behavior is achieved regardless of the structure of a program. The goal is for the programmer to (re)arrange the actors and tasks in a logical way until the desired behavior is achieved. A minimum viable program can be implemented using the "move to" actor alone. This actor displays a small icon reflecting the robot's target pose as at that specific waypoint. After having understood the working principle of actors, one can implement more advanced functionalities.

Besides the sequential orientation of programs, *Assembly* differs from Blockly in the way in which it handles parameters and variables. Whereas Blockly allows users to integrate variables and parameters into blocks, *Assembly* uses a special screen region called *Context* (i.e., lower left-hand side in Figure 2), where parameters and variables can be defined using an embedded JavaScript Object Notation (JSON) editor. The variables and parameters thus defined can be used with control structures, which contain a dropdown list that is automatically populated with compatible variables, and actors (e.g., the *"dP—dR"* relative motion actor). To set parameters or variables before the execution of an actor or task, *Assembly* provides two specialized gray-colored actors, whose icons match those of the "Parameters" and "Variables" buttons in the *Context*, respectively. Following the *Blackboard* design pattern [58], tasks and actors only communicate with each other by reading and writing into a shared JSON object (i.e., the blackboard), which internally represents the *Context* of the current task. Besides the "Parameters" and "Variables" sections, the *Context* also contains a read-only "Robot" section which provides real-time access to the coordinates of the robot during execution and a "Task" section, which allows users to parameterize tasks.

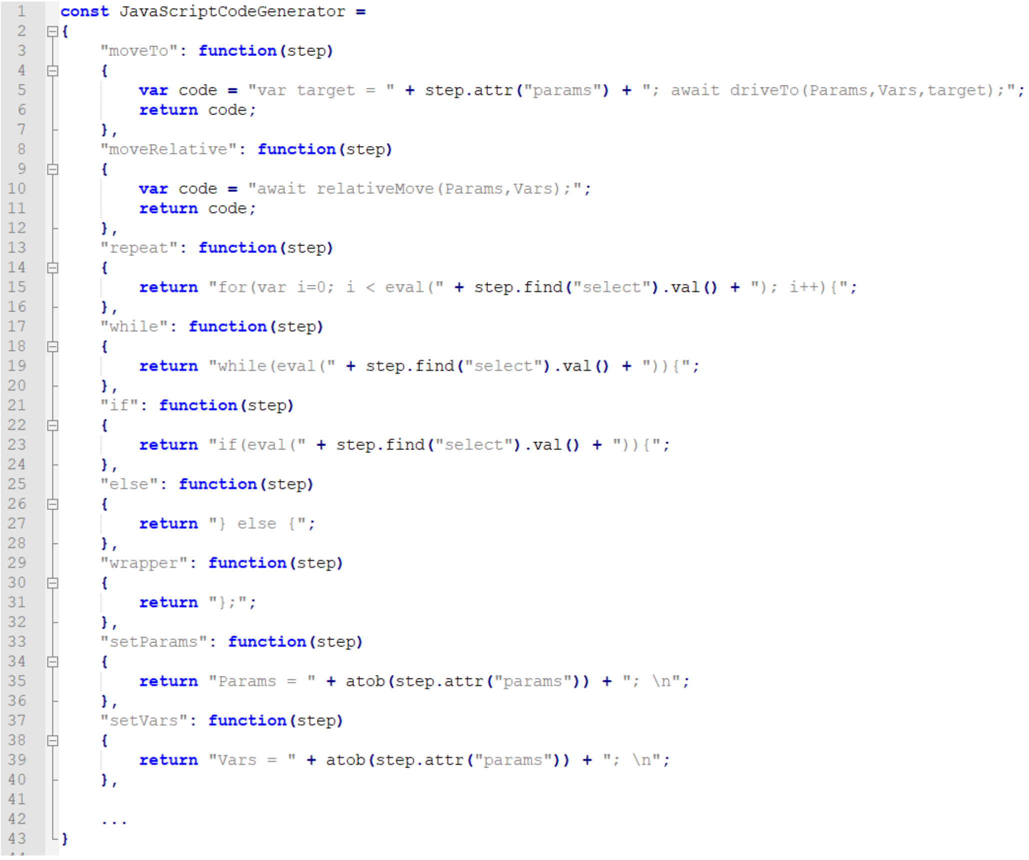
As opposed to *Assembly*, Blockly provides additional types of blocks that have predefined structures and possibly multiple explicit parameters. While this provides custom block creators with more possibilities and flexibility, defining everything (including variables) as a block can, arguably, lead to complicated, non-intuitive constructs for assignments and expressions. Blocks taking many parameters may also expand horizontally or vertically, thus taking up lots of space of the viewport for relatively simple logic. For these reasons, *Assembly* strives to standardize actors in order to simplify their creation and use by unexperienced programmers.

From a structural point of view, there are three main types of actors in *Assembly*:

3. *Non-parameterizable actors*, which do not use any parameters, such as the "else" or "tail" actors;
4. *Implicitly parameterizable actors*, which use variables and parameters from the *Context* but neither explicitly expose them nor require user-defined values for them, such as movement and end effector actors;
5. *Explicitly parameterizable actors*, which expose one variable, the value of which needs to be set by the user in order for the actor to produce a meaningful effect. This is the case for control structures.

By separating the control flow (i.e., workflow) from the data flow (i.e., Context), *Assembly* aims to facilitate an easy entry to novice programmers, while providing a so-called "exit" strategy [31] for more advanced ones. The exit strategy consists in incentivizing users to learn how to work with variables and expressions in the *Context*, which are required by different control structures. At the same time, absolute novices can still create functional workflow using non-parameterizable and implicitly parameterizable actors. Once the logic, composition, structure, and expression format of an *Assembly* program are mastered, the user can move on to textual programming.

By default, *Assembly* generates a JavaScript program, which can be inspected by the user and copied to the clipboard. The basic structure of a language-specific code generator in *Assembly* is that of a JSON object, which provides callback functions as object properties corresponding to a specific actor, as shown in Figure 3. To generate a program, *Assembly* iterates through the current workflow, represented as a list of actors, and calls the function corresponding to the type of the actor being processed in the current iteration. *Assembly's* generic code generation algorithm can thus be easily coupled to other language-specific generators.



```
const JavaScriptCodeGenerator =
{
    "moveTo": function(step)
    {
        var code = "var target = " + step.attr("params") + "; await driveTo(Params,Vars,target);";
        return code;
    },
    "moveRelative": function(step)
    {
        var code = "await relativeMove(Params,Vars);";
        return code;
    },
    "repeat": function(step)
    {
        return "for(var i=0; i < eval(" + step.find("select").val() + "); i++){";
    },
    "while": function(step)
    {
        return "while(eval(" + step.find("select").val() + ")){";
    },
    "if": function(step)
    {
        return "if(eval(" + step.find("select").val() + ")){";
    },
    "else": function(step)
    {
        return "} else {";
    },
    "wrapper": function(step)
    {
        return "};";
    },
    "setParams": function(step)
    {
        return "Params = " + atob(step.attr("params")) + "; \n";
    },
    "setVars": function(step)
    {
        return "Vars = " + atob(step.attr("params")) + "; \n";
    },

    ...
}
```

Figure 3. JavaScript code listing showing an extract from the JavaScript code generator.

3.2. *A Generic Robot Programming Interface (GRPI)*

Figure 4 shows an UML diagram of a GRPI and its relations to the software components that are necessary to generate a robot program from an RPA model. The package RobotHMIAutomation contains a GRPI called IRobotProgramGenerator as well as several implementations of that GRPI, e.g., for UR, Franka Emika Panda, and KUKA iisy robots. The GRPI specifies a series of required fields and methods that need to be provided or implemented by any robot-specific program generator class, module, or component. The dependency of the RobotHMIAutomation package on an external RPA library is explicitly illustrated using a dashed UML arrow. The RPA library used by the RobotHMIAutomation package should at least provide a *click(Image)* and a *type(String)* function. For example, a call of the *click(Image)* function determines the RPA engine to emulate the user action required to perform a left-mouse click upon the graphical element specified by the provided image parameter. These images are specific to the proprietary robot programming environment for which the robot program is being generated. They should thus be considered as being inherent components of any implementation of the GRPI and should be packaged together with that implementation.

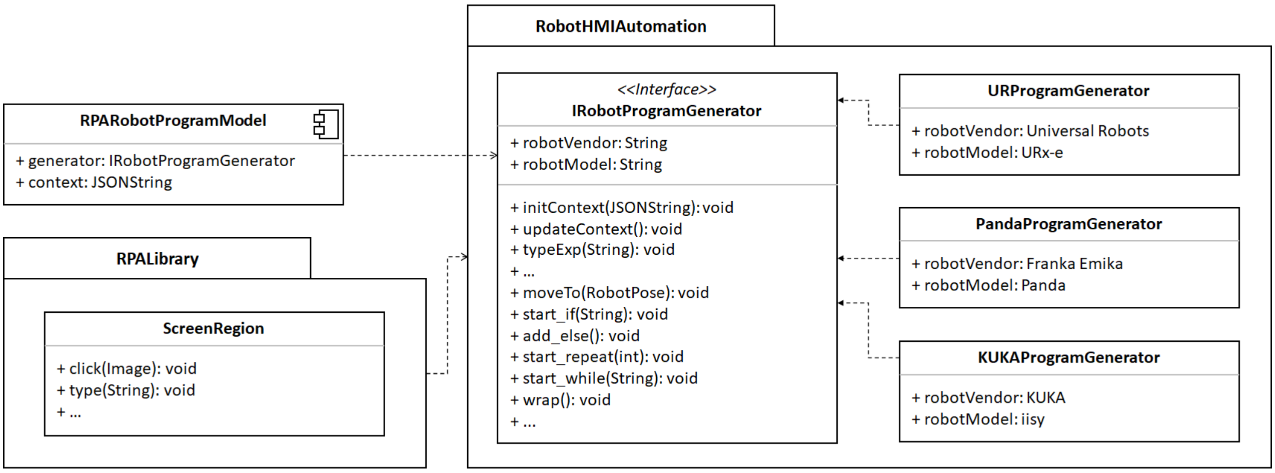
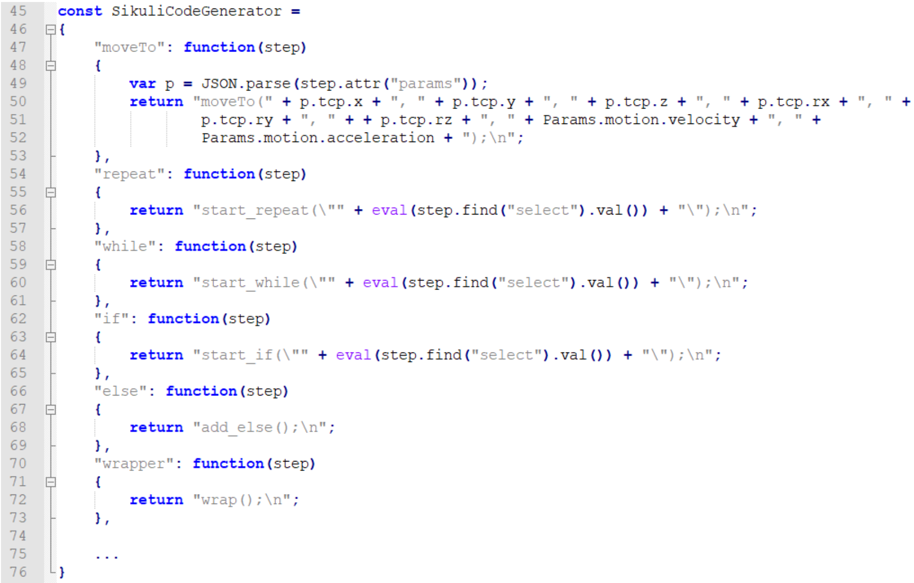


Figure 4. Specification of a GRPI (generic robot programming interface) and its relations to a plurality of robot program generator classes, components, or modules; a generic graphical user interface (GUI) automation (or robotic process automation (RPA)) library; an RPA robot program model.

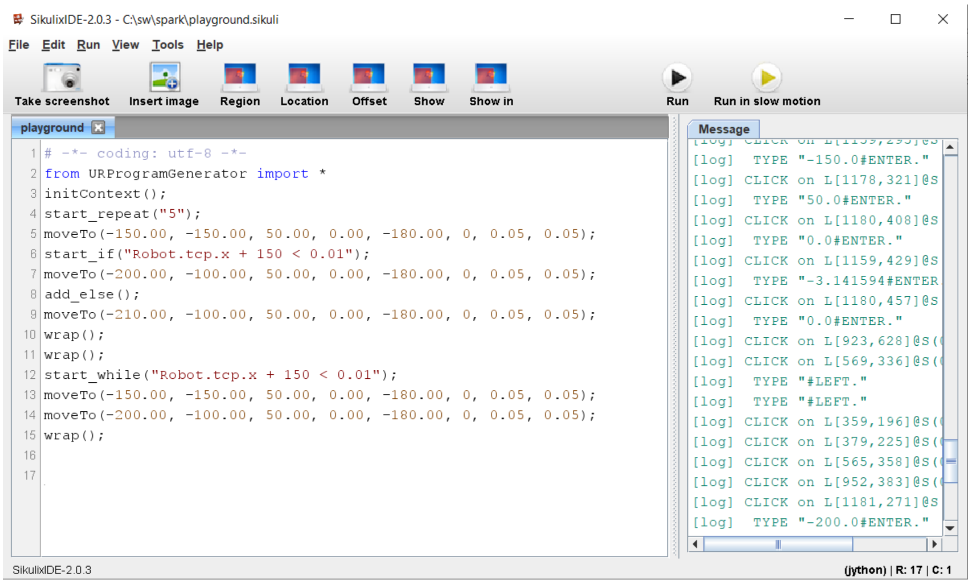
To facilitate the generation of programs in a proprietary target program environment, a GRP environment must implement a generic RPA model generator, which conforms to the *IRobotProgramGenerator* interface. This approach is similar to the Page Object design pattern in GUI automation, which allows wrapping a GUI in an object-oriented way [42]. Figure 5 depicts an extract from the JSON object implementing such a generator in *Assembly*. The *SikuliCodeGenerator* generates RPA models from robot programs created using *Assembly's* graphical editor. These models are designed to work with the Sikuli/SikuliX RPA tool [13,59], which uses Python as its scripting language.



```
const SikuliCodeGenerator =
{
    "moveTo": function(step)
    {
        var p = JSON.parse(step.attr("params"));
        return "moveTo(" + p.tcp.x + ", " + p.tcp.y + ", " + p.tcp.z + ", " + p.tcp.rx + ", " +
                p.tcp.ry + ", " + + p.tcp.rz + ", " + Params.motion.velocity + ", " +
                Params.motion.acceleration + ");\n";
    },
    "repeat": function(step)
    {
        return "start_repeat(\"" + eval(step.find("select").val()) + "\");\n";
    },
    "while": function(step)
    {
        return "start_while(\"" + eval(step.find("select").val()) + "\");\n";
    },
    "if": function(step)
    {
        return "start_if(\"" + eval(step.find("select").val()) + "\");\n";
    },
    "else": function(step)
    {
        return "add_else();\n";
    },
    "wrapper": function(step)
    {
        return "wrap();\n";
    },

    ...
}
```

Figure 5. Code listing showing an extract from the RPA model generator for Sikuli/SikuliX.

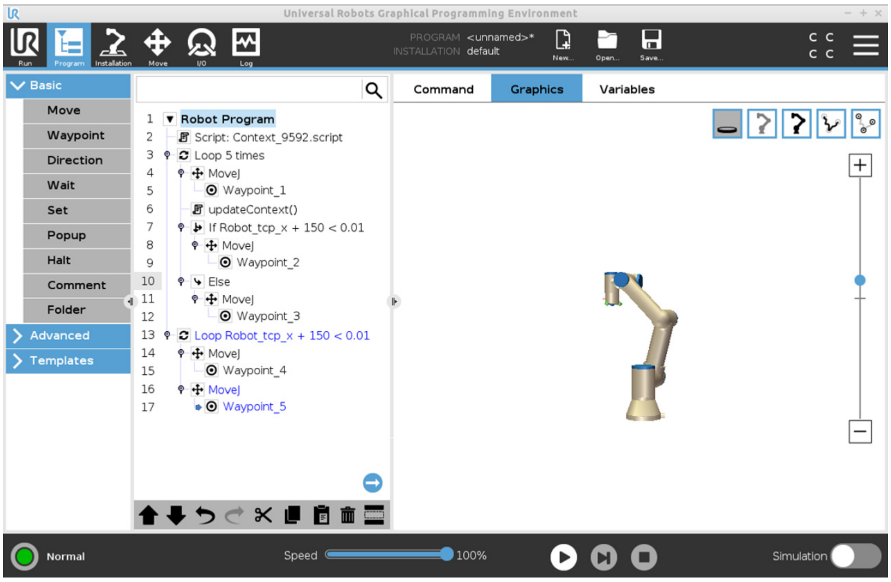
Figure 6 shows the code generated by *Assembly* for the robot program depicted in Figure 1; whereas Figure 7 depicts the program generated in UR's proprietary HMI environment (Polyscope) corresponding to the RPA model from Figure 6. The *wrap()* command in Figure 6 corresponds to a closing accolade (i.e., "}") in a C-style language and the "tail" actors in *Assembly*. Depending on the programming model of the target robot system, this command may, for example, close a conditional structure or a loop by wrapping up the elements pertaining to the respective program structure by generating a terminator element or moving the program cursor to the same level as the corresponding program structure. Hence, in this RPA model as in the GRPI, all control structures are prefixed by *start_*, thus signaling that the command requires a *wrap()* element later on in the program; whereas the subsequent elements of an articulated control structure, such as *if-elseif-else*, will be prefixed by *add_*, as in *add_elseif* and *add_else*. Together with the *wrap()* instruction, these prefixes facilitate the reconstruction of a program's abstract syntax tree, which is a necessary and sufficient condition for generating code or other program semantics in any target programming model. This mechanism is illustrated in the next section in more detail by the example of a *URProgramGenerator* for Universal Robots' Polyscope HMI.



```
# -*- coding: utf-8 -*-
from URProgramGenerator import *
initContext();
start_repeat("5");
moveTo(-150.00, -150.00, 50.00, 0.00, -180.00, 0, 0.05, 0.05);
start_if("Robot.tcp.x + 150 < 0.01");
moveTo(-200.00, -100.00, 50.00, 0.00, -180.00, 0, 0.05, 0.05);
add_else();
moveTo(-210.00, -100.00, 50.00, 0.00, -180.00, 0, 0.05, 0.05);
wrap();
wrap();
start_while("Robot.tcp.x + 150 < 0.01");
moveTo(-150.00, -150.00, 50.00, 0.00, -180.00, 0, 0.05, 0.05);
moveTo(-200.00, -100.00, 50.00, 0.00, -180.00, 0, 0.05, 0.05);
wrap();
```

```
[log] CLICK on L[1159,295]@S
[log]  TYPE "-150.0#ENTER."
[log] CLICK on L[1178,321]@S
[log]  TYPE "50.0#ENTER."
[log] CLICK on L[1180,408]@S
[log]  TYPE "0.0#ENTER."
[log] CLICK on L[1159,429]@S
[log]  TYPE "-3.141594#ENTER
[log] CLICK on L[1180,457]@S
[log]  TYPE "0.0#ENTER."
[log] CLICK on L[923,628]@S(
[log] CLICK on L[569,336]@S(
[log]  TYPE "#LEFT."
[log]  TYPE "#LEFT."
[log] CLICK on L[359,196]@S(
[log] CLICK on L[379,225]@S(
[log] CLICK on L[565,358]@S(
[log] CLICK on L[952,383]@S(
[log] CLICK on L[1181,271]@S
[log]  TYPE "-200.0#ENTER."
```

Figure 6. RPA robot program model to be used with SikuliX. This RPA model was exported from *Assembly* and corresponds to the workflow depicted in Figure 1.



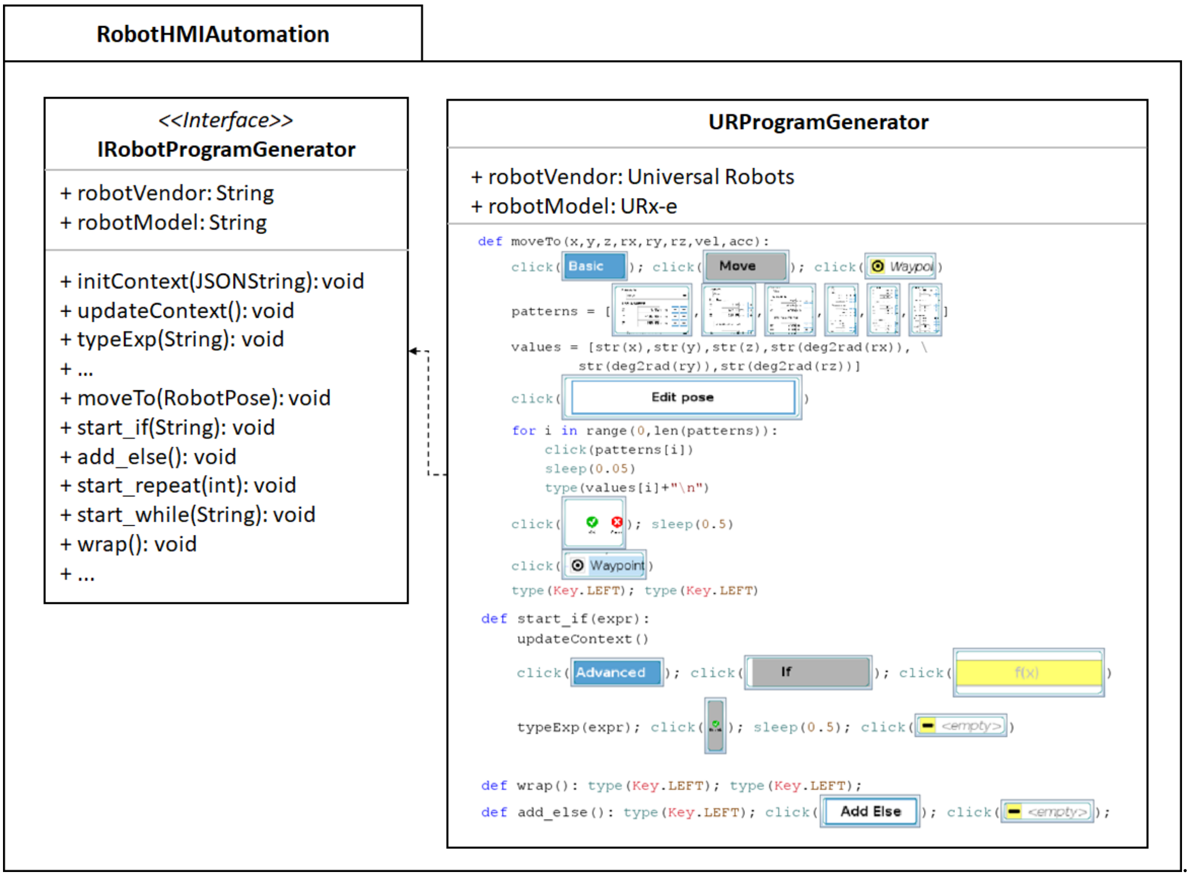
```
▼ Robot Program
    Script: Context_9592.script
    Loop 5 times
        MoveJ
            Waypoint_1
        updateContext()
        If Robot_tcp_x + 150 < 0.01
            MoveJ
                Waypoint_2
        Else
            MoveJ
                Waypoint_3
    Loop Robot_tcp_x + 150 < 0.01
        MoveJ
            Waypoint_4
        MoveJ
            Waypoint_5
```

Figure 7. Robot program generated in Universal Robot's graphical programming environment (Polyscope). The program corresponds to the RPA model from Figure 6 and the *Assembly* workflow from Figure 1.

3.3. Robot-Specific Program Generators

Robot-specific program generators are classes or other kinds of modular software components, which emulate the user interactions required to create the program structures corresponding to the different elements of a GRPI in a target robot's native programming environment (e.g., UR Polyscope, Franka Emika Desk, etc.). These generators must implement the robot independent GRPI, thus bridging a GRP environment such as *Assembly* with robot-specific environments. Robot program generators are analogous to robot drivers in ROS but do not require advanced programming knowledge since they only use a limited number of the functions provided by any modern GUI automation library, such as SikuliX [13] or PyAutoGUI [14].

Figure 8 shows an extract from a *URProgramGenerator*, which implements the *IRobotProgramGenerator* interface and is designed to work with SikuliX. In the graphical user interface of SikuliX, the *click* command, which emulates a left-button click in a designated region (in this example, the designated region is the entire screen) allows for specifying the target GUI element using a visible picture, which is captured directly on screen. A new program generator can thus be created by reconstructing the steps required of a user to create all the program structures specified by the GRPI. For example, generating a motion instruction (i.e., *moveTo*) having a single waypoint requires emulating a series of clicks on different elements of the target robot's HMI (in this example, Universal Robots' Polyscope) and typing the coordinates of the target robot pose in the corresponding text fields in the HMI. In the case of Polyscope, the *wrap()* instruction is implemented by emulating two subsequent presses of the left arrow button on a generic keyboard, which moves the program cursor out of the current control structure.



<<Interface>>
IRobotProgramGenerator

+ robotVendor: String
+ robotModel: String

+ initContext(JSONString): void
+ updateContext(): void
+ typeExp(String): void
+ ...
+ moveTo(RobotPose): void
+ start_if(String): void
+ add_else(): void
+ start_repeat(int): void
+ start_while(String): void
+ wrap(): void
+ ...

URProgramGenerator

+ robotVendor: Universal Robots
+ robotModel: URx-e

```
def moveTo(x,y,z,rx,ry,rz,vel,acc):
    click(Basic); click(Move); click(Waypoi)
    patterns = [ , , , , , ]
    values = [str(x),str(y),str(z),str(deg2rad(rx)), \
          str(deg2rad(ry)),str(deg2rad(rz))]
    click(Edit pose)
    for i in range(0,len(patterns)):
        click(patterns[i])
        sleep(0.05)
        type(values[i]+"\n")
    click( ); sleep(0.5)
    click(Waypoint)
    type(Key.LEFT); type(Key.LEFT)
def start_if(expr):
    updateContext()
    click(Advanced); click(If); click(f(x))
    typeExp(expr); click( ); sleep(0.5); click(<empty>)
def wrap(): type(Key.LEFT); type(Key.LEFT);
def add_else(): type(Key.LEFT); click(Add Else); click(<empty>);
```

Figure 8. Extract from a *URProgramGenerator* designed to work with SikuliX.

The program generator also provides a series of auxiliary functions, including *updateContext* and *typeExp* (not shown in Figure 8 but visible in the generated Polyscope program in Figure 7). The *updateContext* function is used in connection with the *initContext* function specified by the *IRobotProgramGenerator* interface. *initContext* replicates the parameters and variables from the *Context* object in *Assembly* such that all variables available within the scope of the source program be reconstructed in the target program. In the case of Polyscope, *initContext* creates a script command (see line 2 in Figure 7), which creates all the parameters and variables contained in the *Context* object. The *updateContext* function is inserted by the generator before control structures in order to update all variables and parameters (e.g., line 6 in Figure 7).

4. Evaluation

This section presents an evaluation of the proposed approach to GRP on the basis of an application centered on the use of robots in makerspaces. After introducing this application, we describe the infrastructure used in the makerspace to enable the generation of robot programs for the Polyscope HMI of a UR5 robot and discuss some key non-functional requirements, including development effort, program generation performance, and reliability. Then, we present the results of an assessment of the generalizability of the approach based on the creation and analysis of program generators for two additional program models that are commonly used in robot programming (i.e., block-based and list-based programming), which differ from the tree-based model used in Polyscope. Finally, we discuss the benefits and limitations of the proposed approach based on the results of this evaluation.

4.1. Application: Robotics in Makerspaces

Makerspaces are shared machine shops in which people that have diverse backgrounds are provided with access to advanced manufacturing technologies, which typically include 3D printers, laser cutters, and other additive and subtractive manufacturing systems. Makerspace members, who usually do not have specific institutional affiliations, can use these technologies for their own creative, educational, or work-related purposes, while paying a membership fee. With the recent advances in collaborative robotics, more robots are being encountered in European makerspaces than ever before. This attracts robot vendors that are increasingly interested in non-industrial markets as well as companies and research institutions interested in leveraging the open innovation potential in makerspaces [17]. In this sense, makerspaces pose new safety challenges, since member applications are not usually known in advance. In this respect, makerspaces differ from traditional industrial contexts, where robot applications must undergo safety certifications before they can be used productively [17].

Since the number of robots in any makerspace is limited and machine hours may cost additionally, makerspace members can benefit from learning and exercising generic robot programming at home using a web-based tool such as *Assembly* before testing the programs on the robots in the makerspace. Sketching and sharing an idea with other members over a web-based robot programming environment also helps to identify potential safety issues entailed by an application. These mostly autodidactic, preparatory practices of web-based programming are complemented by the robotics trainings offered by makerspaces, which emphasize safety but usually do not go into great details about programming, since not all trainees have programming experience. Acquiring a basic background in robot programming at home thus enables members to benefit from more advanced programming trainings as well as from asking robotics trainers more precise and purposeful questions, grounded in the goals and needs of the robotics projects they envision. This arguably fosters a problem-oriented, constructivist learning environment [60,61] in the makerspace, while the absolute basics are self-taught. In the makerspace, members can then translate their robot programs for the different available robots, for example, to determine which of them may be best suited for the task at hand. At the same time, using the GRPI, robotics trainers can develop program generators for the different robot types available in the makerspace.

Safety is a challenging aspect in makerspace robotics, since—as opposed to an industrial context—makerspace member applications are not known in advance and therefore cannot be certified [17,18]. Therefore, makerspace members are responsible for their own safety [17], whereas makerspace representatives need only configure the safety setting in the robots' native HMIs to ensure member safety (e.g., by enforcing strict, password-protected speed and force limits). Makerspace members can thus generate programs for any available robot without losing the safety-related and other features provided by the robot's native HMI. In addition, the task of translating a generic source program into a robot-specific target one also stimulates users to learn how to work with different robots and programming environments by example.

Figure 9 illustrates the usage scenario and infrastructure that can be used in the makerspace to support program creation in *Assembly* at home and target program generation for a typical robotic system (in this example, a UR5), composed of a robot arm, a control computer, and a teach pendant hosting the HMI. To connect the toolchain used for program generation (i.e., *Assembly*–SikuliX–Robot HMI), the HMI software hosted on the physical teaching pendant is mirrored (or replicated) using a secure wired connection (e.g., USB, Ethernet) and a remote monitoring and control tool (in this example, RealVNC) running on a computer provided by the makerspace. Universal Robots allows mirroring the robot's HMI using a VNC server. Other vendors (e.g., KUKA, ABB, Fanuc) offer similar open or proprietary solutions for remotely monitoring and controlling a robot's HMI in operation. The makerspace computer hosts the SikuliX GUI automation tool, which also provides robot-specific program generators. This way, members can generate programs in a target environment over a remote connection to the robot's HMI. The setup in Figure 9 also ensures that members can work safely with collaborative robots by using the safety features of the robot.

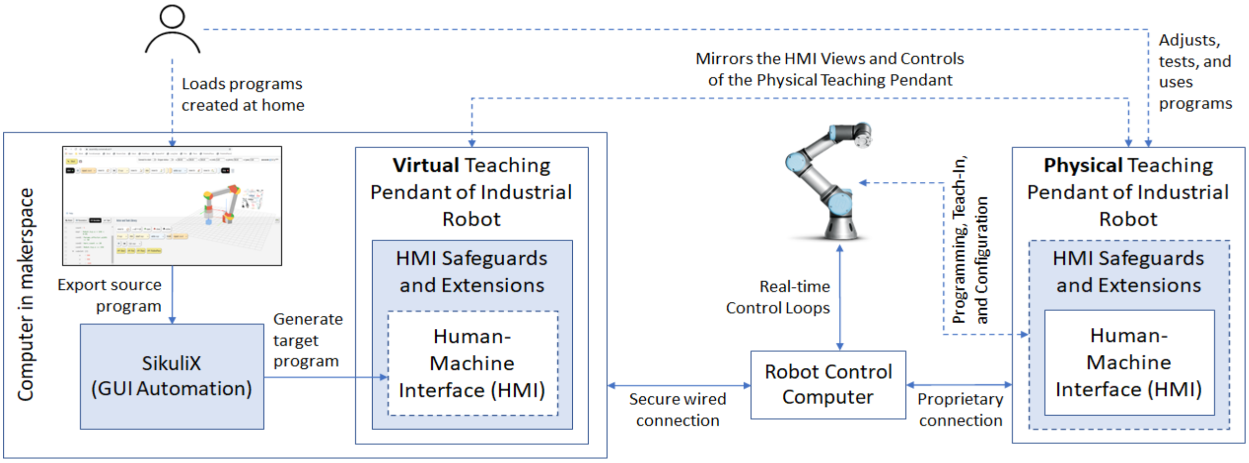

Loads programs created at home
Mirrors the HMI Views and Controls of the Physical Teaching Pendant
Adjusts, tests, and uses programs
Computer in makerspace
Export source program
SikuliX (GUI Automation)
Generate target program
Virtual Teaching Pendant of Industrial Robot
HMI Safeguards and Extensions
Human-Machine Interface (HMI)
Real-time Control Loops
Programming, Teach-In, and Configuration
Robot Control Computer
Secure wired connection
Proprietary connection
Physical Teaching Pendant of Industrial Robot
HMI Safeguards and Extensions
Human-Machine Interface (HMI)

Figure 9. Usage scenario and infrastructure for GRP and program generation used in a Makerspace.

Table 1 shows some key performance indicators of program generation using the setup from Figure 9. Due to the latency of Polyscope, to which around 100 ms are added by the remote connection, and around 25 ms by the image recognition algorithms from the OpenCV library [62], which is used by SikuliX, the program generation is not as fast as when using the URSim on a powerful computer (e.g., Intel i7 or equivalent with 16GB RAM). The overall difference between the two setups in terms of total program generation time is low because the generator uses a feed-forward strategy—i.e., it does not wait for a success signals, which allows it to pipeline the generation of instructions. The relatively slow performance of the program generation process can also be turned into a useful feature, which allows users to configure the speed of program generation in a target environment to fit their learning pace.

Table 1. Key performance indicators of program generation for the Universal Robots' model 5 (UR5) robot.

Indicator	UR5 Robot	URSim
HMI Latency	110 ms	40 ms
Remote connection/VM latency	100 ms	15 ms [1]
Processing time per GUI element	25 ms	25 ms
Total program generation time	58 s	48 s

[1] URSim runs in an Ubuntu virtual machine (VM), which induces some latency.

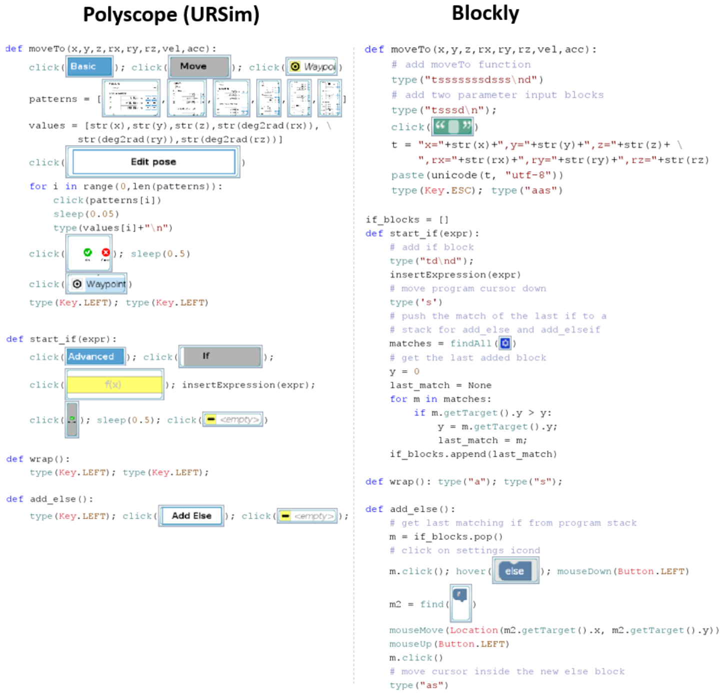
4.2. Generalizability

To assess the generalizability of the approach to other programming models, two additional program generators were created for Blockly and a proprietary list-based programming environment, called Robot Control [37], which is provided with IGUS®robots. The rationale for this choice was twofold. First, block-based robot programming environments, which are provided with many educational (e.g., Dobot, UArm, Niryo, etc.) and lightweight collaborative robots (e.g., ABB single-arm Yumi [63]), are increasingly popular because they facilitate an easy entry into robotics programming. Similarly, list-based programming environments represent a popular "no-code" alternative to textual robot programming. For example, modern web-based generic programming environments, such as Drag&Bot, build on list-based programming. Second, simplified block-based and list-based environments usually do not provide means for importing (i.e., converting) programs created using other tools.

The evaluation procedure was as follows. First, a simple test program was created in *Assembly* and exported as an RPA model for the SikuliX environment. The content of that RPA model was:

```
start_if("a==b")
moveTo(280,185,250,-150,-7,-180,50,60)
add_else();
start_if("c==a")
moveTo(260,185,250,-150,-7,-180,50,60)
wrap()
wrap()
start_if("c==b")
moveTo(270,185,250,-150,-7,-180,50,60)
wrap()
```

The aim with this program was to measure the implementation effort required to create generators that are able to produce (1) motion statements and (2) control structures in diverse target programming environments, and to assess the performance and complexity of those generators when applied to the test program listed above. Figure 10 shows extracts from the GRPI-conforming generators corresponding to the three robot programming environments that were used in the evaluation (UR Polyscope, Blockly, and IGUS®Robot Control); whereas Figure 11 shows the programs they produce in the respective environments. The measured key indicators are presented in Table 2.



Polyscope (URSim)

```
def moveTo(x,y,z,rx,ry,rz,vel,acc):
    click(Basic); click(Move); click(⊙ Waypoi)
    patterns = [ , , , , , ]
    values = [str(x),str(y),str(z),str(deg2rad(rx)), \
              str(deg2rad(ry)),str(deg2rad(rz))]
    click(Edit pose)
    for i in range(0,len(patterns)):
        click(patterns[i])
        sleep(0.05)
        type(values[i]+"\n")
    click(✓ ✗); sleep(0.5)
    click(⊙ Waypoint)
    type(Key.LEFT); type(Key.LEFT)


def start_if(expr):
    click(Advanced); click(If);
    click(f(x)); insertExpression(expr);
    click(); sleep(0.5); click(▬ <empty>)

def wrap():
    type(Key.LEFT); type(Key.LEFT);

def add_else():
    type(Key.LEFT); click(Add Else); click(▬ <empty>);
```

Blockly

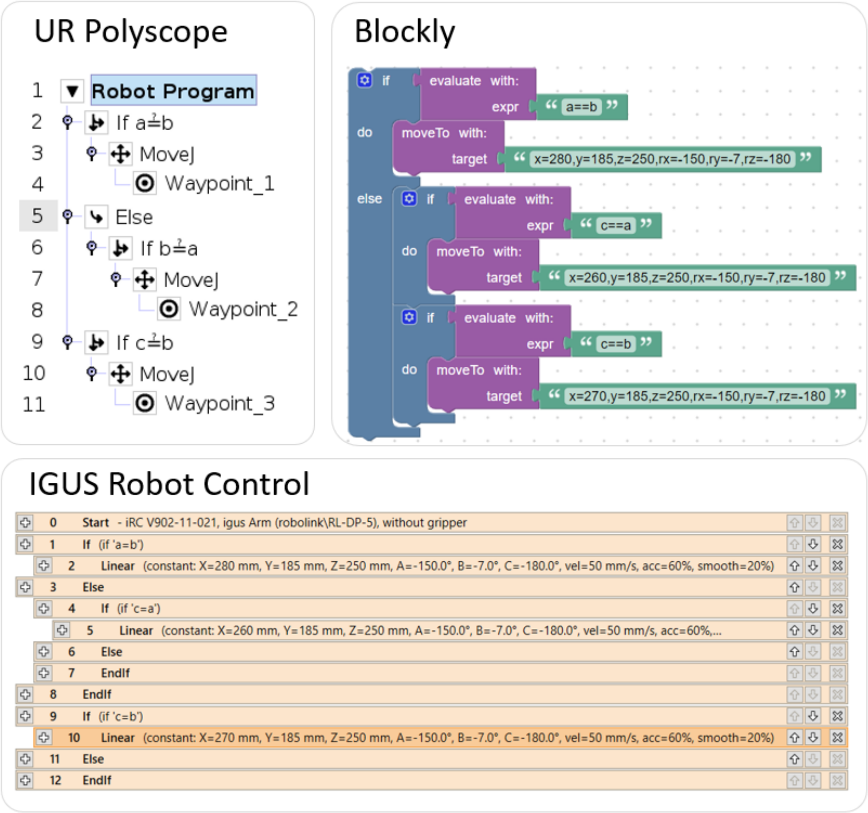
```
def moveTo(x,y,z,rx,ry,rz,vel,acc):
    # add moveTo function
    type("tsssssssdsss\nd")
    # add two parameter input blocks
    type("tsssd\n");
    click(“ ”)
    t = "x="+str(x)+",y="+str(y)+",z="+str(z)+ \
        ",rx="+str(rx)+",ry="+str(ry)+",rz="+str(rz)
    paste(unicode(t, "utf-8"))
    type(Key.ESC); type("aas")

if_blocks = []
def start_if(expr):
    # add if block
    type("td\nd");
    insertExpression(expr)
    # move program cursor down
    type('s')
    # push the match of the last if to a
    # stack for add_else and add_elseif
    matches = findAll(⚙)
    # get the last added block
    y = 0
    last_match = None
    for m in matches:
        if m.getTarget().y > y:
            y = m.getTarget().y;
            last_match = m;
    if_blocks.append(last_match)

def wrap(): type("a"); type("s");

def add_else():
    # get last matching if from program stack
    m = if_blocks.pop()
    # click on settings icond
    m.click(); hover(else); mouseDown(Button.LEFT)

    m2 = find( )

    mouseMove(Location(m2.getTarget().x, m2.getTarget().y))
    mouseUp(Button.LEFT)
    m.click()
    # move cursor inside the new else block
    type("as")
```

IGUS® Robot Control

```
def moveTo(x,y,z,rx,ry,rz,vel,acc):
    click(Action); click(⚡ Linear motion);
    patterns = [Velocity:, Acceleration:, X:, Y:, Z:, A:, B:, C:]
    offsets = [50,50,30,30,30,30,30,30]
    values = [vel,acc,x,y,z,rx,ry,rz]

    for i in range(0,len(patterns)):
        hover(patterns[i])
        m = getLastMatch()
        click(Location(m.getTarget().x + offsets[i], m.getTarget().y))
        type(str(values[i]))

    click( )

stack = []
def start_if(expr):
    stack.append("start_if")
    click(Flow); click(⚡ If...then...else);
    doubleClick(Condition True); insertExpression(expr);
    click( )

def wrap():
    if len(stack) > 0 and stack.pop() == "add_else":
        type(Key.DOWN)
    elif len(stack) > 0:
        type(Key.DOWN);type(Key.DOWN)

def add_else():
    stack.append("add_else")
    type(Key.DOWN)
```

Figure 10. Comparison of the key functions extracted from the corresponding robot-specific program generators (implemented in Python within SikuliX), which were used to generate the test program in three different programming environments.

First, it must be noted that it was technically possible to create program generators for each of the target environments being tested with relatively little development effort. The effort required to create these generators differed, with Blockly yielding the highest value. The reason for this is that Blockly does not use a fixed program pane, like Polyscope and Robot Control, which require users to create program elements at exact locations below preceding statements or within different control statements using instruction-specific control buttons or menu items. Therefore, Blockly's keyboard navigation feature was used whenever it provided a simpler solution than emulating the dragging and dropping of elements using mouse control. In Blockly, keyboard control requires a certain amount of time to learn and master. In addition, although the length of the Blockly generator is approximately equal to the lengths of the other two, the relatively long combination of keys required to navigate through the menus must be accompanied by more elaborate comments in the code of the generator. Additionally, the generator functions for the *if-then-else* statement in Blockly are sensibly more complex, which is an effect of the keyboard navigation and the mechanism by which parameters are associated with different blocks. It must also be emphasized that generation of native Blockly expressions is cumbersome, which prompted the decision to implement a helper function that evaluates expressions provided as strings rather than "spelling out" native Blockly expressions. Although with enough ingenuity a patient programmer could implement an RPA model able to generate native Blockly expressions, this was beyond the scope of this evaluation. In this sense, it suffices to note that each (graphical) programming model uses (slightly) different expression syntax and semantics, which requires appropriate solutions or workarounds to enable RPA-based program generation. The choice of keyboard navigation over drag-and-drop programming may be regarded as a design tactic, which can be applied whenever emulating keyboard rather than mouse interactions is more effective.



UR Polyscope

```
▼ Robot Program
  If a≟b
    MoveJ
      Waypoint_1
  Else
    If b≟a
      MoveJ
        Waypoint_2
  If c≟b
    MoveJ
      Waypoint_3
```

Blockly

```
if evaluate with:
       expr "a==b"
do   moveTo with:
       target "x=280,y=185,z=250,rx=-150,ry=-7,rz=-180"
else if evaluate with:
          expr "c==a"
     do   moveTo with:
          target "x=260,y=185,z=250,rx=-150,ry=-7,rz=-180"
     if evaluate with:
          expr "c==b"
     do   moveTo with:
          target "x=270,y=185,z=250,rx=-150,ry=-7,rz=-180"
```

IGUS Robot Control

```
0   Start - iRC V902-11-021, igus Arm (robolink\RL-DP-5), without gripper
1   If (if 'a=b')
 2    Linear (constant: X=280 mm, Y=185 mm, Z=250 mm, A=-150.0°, B=-7.0°, C=-180.0°, vel=50 mm/s, acc=60%, smooth=20%)
3   Else
 4    If (if 'c=a')
  5     Linear (constant: X=260 mm, Y=185 mm, Z=250 mm, A=-150.0°, B=-7.0°, C=-180.0°, vel=50 mm/s, acc=60%,...
 6    Else
 7    EndIf
8   EndIf
9   If (if 'c=b')
 10   Linear (constant: X=270 mm, Y=185 mm, Z=250 mm, A=-150.0°, B=-7.0°, C=-180.0°, vel=50 mm/s, acc=60%, smooth=20%)
11  Else
12  EndIf
```

Figure 11. Generated programs in (1) UR Polyscope, (2) Blockly, and (3) IGUS®Robot Control corresponding to the test program.

Table 2. Key performance indicators of program generation for the UR5 robot.

Indicator	Polyscope (URSim)	Blockly	Robot Control
Repeatability	100%	100%	100%
Implementation effort	~5 h	~8 h	~4 h
Complexity of generator	low	medium	low
Generation time for test program	38.5 s [1]	11.6 s	22.5 s

[1] URSim runs in an Ubuntu virtual machine (VM), which induces some latency.

The results also suggest that list-based (i.e., Robot Control) and tree-based (i.e., Polyscope) programming are easier to emulate using an RPA approach than block-based programming. This is likely because, in Blockly, *Assembly*, and other similar environments, programs and statements are represented as sequences of blocks of different sizes and shapes; whereas in Polyscope and Robot Control, the structure of statements is rather fixed and more easily navigable, either by keyboard or by mouse. This is unsurprising since Blockly's aim is to facilitate a learner's transition towards textual programming and thus to foster creativity. By contrast, classic robot programming models are more straightforward and do not have an exit strategy—i.e., a strategy to facilitate the user's transition towards textual programming [31].

Concerning the reliability of RPA-based program generation, repeatability arguably provides a good indicator. The program generation process was repeated 50 times for each target environment and, once a working setup was achieved, the OpenCV image recognition algorithms integrated in SikuliX never failed on a 1920 × 1080 screen.

In terms of performance, thanks to the keyboard navigation feature, the Blockly generator outperformed the other two generators. The relatively slow performance of the

Polyscope generator can be explained by the fact that URSim (i.e., the Polyscope simulation environment) runs in an Ubuntu virtual machine and uses a rather slow Java UI framework, which induces some latency. For example, Polyscope uses modal dialogues for user inputs (e.g., target locations), which take approximately 100 ms to pop up in the virtual machine.

5. Discussion

Overall, the evaluation results suggest that the proposed approach to generic robot programming by emulating user interactions in an entire range of graphical (block, tree, or list-based) target robot programming environments. Textual robot programming environments were not evaluated as targets since code generation is a well-studied and easily solvable problem. In addition, *Assembly* generates RPA models in Python, which demonstrates the systems' capability to generate GRPI-conforming code. The evaluation suggests that there are several usage scenarios in which the proposed approach presents advantages over existing GRP approaches.

In a first scenario, the proposed approach can be used to generate programs in proprietary graphical robot programming environments, which do not allow importing programs created using other tools. This is the case both for IGUS®Robot Control and Blockly (which is used for example with ABB's one-armed Yumi robot). With such environments, users are provided with no other option than to recreate existing programs whenever the latter needs to be ported to a new robot, which uses a proprietary programming model. This category of robots also includes the newest generation of low cost cobots, such as—for example—the Franka Emika Panda robot, which uses a proprietary web-based programming environment.

In a second scenario, the members of shared machine shops, such as makerspaces, can benefit from the approach by creating robot programs at home and converting them for an entire range of target programming environments in the makerspace. This helps to democratize robot technology by providing inexpensive online access to simplified robot programming, which encourages people who do not typically have access to robots to explore the potential and limits of this technology.

In a third scenario, the automated generation of programs in a target graphical environment fosters learning how to program in the respective target environment. Tools such as SikuliX provide means for easily configuring the pace at which user interactions are emulated (e.g., by configuring the "mouse move delay" parameter). This way, program generation can be adapted to the learning pace of the learner. This also fosters learning to program by example, whereby the examples must be adjusted to fit the requirements of a particular application.

For experts, this way of programming robots may be regarded as a form of test-driven development, in which an application is defined by a series of test cases that need to be passed. In this context, the proposed approach can be used to generate program stubs, which only require the adjustment of locations and waypoints. Generated programs may thus be regarded as design patterns or robot skills, which can be configured and tested on the shop floor.

5.1. Relationship with Component-Based and Model-Driven Robotics Software Engineering

The approach to GRP by GUI automation introduced in this paper shares some similarities with existing component-based [64] and model-driven [65] robotics software engineering approaches. Designing component-based robot software entails mapping various cohesive functionalities to components in order to separate design concerns and produce reusable software building blocks [64]. Furthermore, reusable components should be hardware-independent and interoperable with different software development technologies and control paradigms [64]. In this sense, the robot program generators introduced in Section 3.3 help to abstract dependencies on robot-specific hardware and software, thus enabling the development of reusable components on top of a common GRPI. Nevertheless, imposing restrictions upon the architecture of components is beyond the scope of the approach. Whereas in autonomous robotic applications, componentization and object-orientation help to produce reusable robot software, in practical contexts (e.g., on the factory shop floor and in makerspaces) and in human-robot collaboration applications, developers are confronted with safety, interoperability, and integration issues against the background of time pressure, resource scarcity, and lack of software engineering expertise. In this context, the proposed approach provides a frugal, yet effective solution to implementing result-oriented robot-independent applications in an agile and pragmatic way. At the same time, for users who are more interested in robotics research and advanced, (semi-)autonomous applications, the GRPI provides a required interface and contract that can be used by components, as described in [64].

The proposed approach fully supports model-driven robot programming and software engineering [65]. Within this approach, a domain-specific language and/or modeling environment is used to derive an abstract representation of a real system or phenomenon [65]. In practical terms, the application logic is represented using textual and/or visual semantics, whereby the result is regarded as an (executable) model. Block-based programming environments, such as Blockly and Assembly, are good examples of model-based software authoring tools. Other notable examples include BPMN (Business Process Modeling Notation), which has been used to model and simulate multi-robot architectures [66] and to represent manufacturing processes [67], and the VDI Norm 2680 [27], which has been used to create a constraint-based programming system for industrial robot arms [26]. In this context, the GRPI can be used to interface various modeling languages and environments with robot-specific programming environments. RPA models and robot code generators enable the inclusion of a wide range of robots into high level model representations with relatively little efforts.

5.2. Limitations and Future Work

Despite the successful development of generators for three dissimilar programming models, users should expect some difficulties if keyboard navigation is not supported in some target environments. In such cases, generating some instructions may not be possible without complex workarounds, which might render the image recognition process less reliable. In this sense, evaluation studies should be conducted with more diverse target environments, which heavily rely on mouse or touch-based control.

The relatively long time taken by all generators suggests that users should take into consideration possible delays due to the need of generating target programs repeatedly. In scenarios where program generation needs to be carried out in (near) real time automatically, the approach might thus be inappropriate. The performance of RPA-based program generation is also contingent on the latency of the target environment and will never be as fast as generating textual programs.

Although the time required to develop program generators is comparable to that of developing any other code generator, the proposed approach facilitates test driven development of generators by requiring developers to code in small increments while visually inspecting the result of the generator in the target environment feature by feature. Compared to developing a ROS driver, the required software engineering expertise and development time (including design, code instrumentation, and testing) of program generators is low. However, developing a program generator for complex environments, such as Blockly, requires ingenuity and computational thinking. In addition, as opposed to official APIs, there currently exists no specialized technical support or documentation about how to automate the HMIs of different robots. In this context, the target system's user manual can provide a valuable source of information. Regarding the means for integrating a generated robot program with other systems using GUI automation technique, details are provided in [18].

Whereas in non-industrial, educational contexts, the approach presents several advantages over other approaches (e.g., robotic middleware and robot programming using a teach pendant), an assessment of the utility of the approach in industrial contexts is still needed. Such an assessment could, for example, focus on program generation for legacy robot HMIs. Finally, a comprehensive evaluation of the approach with a significant number of users (with and without an industrial background) focused on the usability of the *Assembly* tool was not yet carried out and is currently in planning.

Supplementary Materials: The source code of the *Assembly* tool is available under the Apache 2.0 license at: https://github.com/CoMeMak/assembly.

Funding: This research was supported by the Austrian Research Promotion Agency (FFG) under Grant number 871459 (https://www.comemak.at).

Institutional Review Board Statement: Not applicable.

Informed Consent Statement: Not applicable.

Conflicts of Interest: The authors declare no conflict of interest.

References

1. Biggs, G.; MacDonald, B. A survey of robot programming systems. In Proceedings of the Australasian Conference on Robotics and Automation, Brisbane, Australia, 10 December 2003.
2. Archibald, C.; Petriu, E. Skills-oriented robot programming. In Proceedings of the International Conference on Intelligent Autonomous Systems IAS-3, Pittsburgh, PA, USA, 15–18 February 1993; pp. 104–115.
3. Freund, E.; Luedemann-Ravit, B. A system to automate the generation of program variants for industrial robot applications. In Proceedings of the IEEE/RSJ International Conference on Intelligent Robots and Systems, Institute of Electrical and Electronics Engineers (IEEE), New York, NY, USA, 30 September–4 October 2002; Volume 2, pp. 1856–1861.
4. Steinmetz, F.; Wollschlager, A.; Weitschat, R. RAZER—A HRI for Visual Task-Level Programming and Intuitive Skill Parameterization. *IEEE Robot. Autom. Lett.* **2018**, *3*, 1362–1369. [CrossRef]
5. Quigley, M.; Conley, K.; Gerkey, B.; Faust, J.; Foote, T.; Leibs, J.; Ng, A.Y. ROS: An open-source Robot Operating System. In Proceedings of the ICRA Workshop on Open Source Software, Kobe, Japan, 25 January 2009; Volume 3, p. 5.
6. RoboDK. Available online: https://robodk.com/ (accessed on 25 November 2020).
7. Siemens Simatic Robot Integration. Available online: https://new.siemens.com/global/en/products/automation/industry-software/automation-software/tia-portal/highlights/robot-integration.html (accessed on 25 November 2020).
8. Drag&Bot. Available online: https://www.dragandbot.com/ (accessed on 25 November 2020).
9. García, S.; Strüber, D.; Brugali, D.; Berger, T.; Pelliccione, P. Robotics software engineering: A perspective from the service robotics domain. In Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Association for Computing Machinery, New York, NY, USA, 8–13 November 2020; pp. 593–604.
10. Wienke, J.; Wrede, S. Results of the survey: Failures in robotics and intelligent systems. *arXiv* **2017**, arXiv:1708.07379.
11. Kolak, S.; Afzal, A.; Le Goues, C.; Hilton, M.; Timperley, C.S. It takes a village to build a robot: An empirical study of the ros ecosystem. In Proceedings of the IEEE International Conference on Software Maintenance and Evolution (ICSME), Institute of Electrical and Electronics Engineers (IEEE), New York, NY, USA, 22 May 2020; pp. 430–440.
12. Estefo, P.; Simmonds, J.; Robbes, R.; Fabry, J. The Robot Operating System: Package reuse and community dynamics. *J. Syst. Softw.* **2018**, *151*, 226–242. [CrossRef]
13. SikuliX. Available online: http://sikulix.com/ (accessed on 25 November 2020).
14. PyAutoGUI. Available online: https://pyautogui.readthedocs.io/ (accessed on 25 November 2020).
15. Bächle, M.; Kirchberg, P. Ruby on rails. *IEEE Softw.* **2007**, *24*, 105–108. [CrossRef]
16. GRAND GARAGE. Available online: https://grandgarage.eu/ (accessed on 25 November 2020).
17. Ionescu, T.B. Meet Your Personal Cobot, But Don't Touch It Just Yet. In Proceedings of the 29th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN), New Delhi, India, 14–18 October 2019; pp. 1113–1118.
18. Ionescu, T.B.; Fröhlich, J.; Lachenmayr, M. Improving Safeguards and Functionality in Industrial Collaborative Robot HMIs through GUI Automation. In Proceedings of the 25th IEEE International Conference on Emerging Technologies and Factory Automation (ETFA), Vienna University of Technology (TU Wien), Vienna, Austria, 8–11 September 2020; Volume 1, pp. 07–564.
19. UR Polyscope. Available online: https://www.universal-robots.com/ (accessed on 25 November 2020).
20. Grover, S.; Pea, R. Computational thinking in K–12: A review of the state of the field. *Educ. Res.* **2013**, *42*, 38–43. [CrossRef]
21. Franka Emika Desk. Available online: https://www.franka.de/apps/ (accessed on 25 November 2020).
22. Ionescu, T.B. When software development meets the shopfloor: The case of industrial fablabs. In Proceedings of the 2019 IEEE/ACM 41st International Conference on Software Engineering: Companion Proceedings (ICSE-Companion), Montreal, QC, Canada, 25–31 May 2019; pp. 246–247.
23. Rethink Robotics Intera. Available online: https://www.rethinkrobotics.com/intera/ (accessed on 25 November 2020).

24. Huang, J.; Cakmak, M. Code3: A system for end-to-end programming of mobile manipulator robots for novices and experts. In Proceedings of the 2017 ACM/IEEE International Conference on HumanRobot Interaction, Vienna, Austria, 6–9 March 2017; pp. 453–462.
25. Hangl, S.; Mennel, A.; Piater, J. A novel skill-based programming paradigm based on autonomous playing and skill-centric testing. *arXiv* **2017**, arXiv:1709.06049.
26. VDI-Richtlinie 2860. *Montage-und Handhabungstechnik; Handhabungsfunktionen, Handhabungseinrichtungen; Begriffe, Definitionen, Symbole*; VDI-Gesellschaft Produktion und Logistik: Düsseldorf, Germany, 2010.
27. Halt, L.; Nagele, F.; Tenbrock, P.; Pott, A. Intuitive Constraint-Based Robot Programming for Robotic Assembly Tasks. In Proceedings of the 2018 IEEE International Conference on Robotics and Automation (ICRA), Institute of Electrical and Elec-tronics Engineers (IEEE), Brisbane, QLD, Australia, 21–25 May 2018; pp. 520–526.
28. Dobot. Available online: https://www.dobot.cc/ (accessed on 25 November 2020).
29. Weintrop, D.; Afzal, A.; Salac, J.; Francis, P.; Li, B.; Shepherd, D.C.; Franklin, D. Evaluating CoBlox: A comparative study of robotics programming environments for adult novices. In Proceedings of the 2018 Conference on Human Factors in Computing Systems (CHI), Montreal, QC, Canada, 21–26 April 2018; pp. 1–12.
30. Mateo, C.; Brunete, A.; Gambao, E.; Hernando, M. Hammer: An Android based application for end-user industrial robot programming. In Proceedings of the IEEE/ASME 10th International Conference on Mechatronic and Embedded Systems and Applications (MESA), Institute of Electrical and Electronics Engineers (IEEE), New York, NY, USA, 10–12 September 2014; pp. 1–6.
31. Fraser, N. Ten things we've learned from Blockly. In Proceedings of the 2015 IEEE Blocks and Beyond Workshop (Blocks and Beyond), Atlanta, GA, USA, 22 October 2015; pp. 49–50.
32. Connolly, C. Technology and applications of ABB RobotStudio. *Ind. Robot. Int. J.* **2009**, *36*, 540–545. [CrossRef]
33. Ionescu, T.B.; Schlund, S. A Participatory Programming Model for Democratizing Cobot Technology in Public and Industrial Fablabs. *Procedia CIRP* **2019**, *81*, 93–98. [CrossRef]
34. Diprose, J.; MacDonald, B.; Hosking, J.; Plimmer, B. Designing an API at an appropriate abstraction level for programming social robot applications. *J. Vis. Lang. Comput.* **2017**, *39*, 22–40. [CrossRef]
35. Lewis, R. *Modeling Distributed Control Systems Using IEC 61499—Applying Function Blocks to Distributed Systems*; IEEE Pub-lishing: New York, NY, USA, 2001.
36. Coronado, E.; Mastrogiovanni, F.; Indurkhya, B.; Venture, G. Visual Programming Environments for End-User Development of intelligent and social robots, a Systematic Review. *J. Comput. Lang.* **2020**, *58*, 100970. [CrossRef]
37. IGUS Robot Control. Available online: https://www.igus.eu/info/robot-software (accessed on 22 November 2020).
38. Eugster, P.T.; Felber, P.A.; Guerraoui, R.; Kermarrec, A.M. The many faces of publish/subscribe. *ACM Comput. Surv.* **2003**, *35*, 114–131. [CrossRef]
39. SMACH. Available online: http://wiki.ros.org/smach (accessed on 17 December 2020).
40. Cervera, E. Try to Start It! The Challenge of Reusing Code in Robotics Research. *IEEE Robot. Autom. Lett.* **2018**, *4*, 49–56. [CrossRef]
41. Artiminds. Available online: https://www.artiminds.com/ (accessed on 25 November 2020).
42. Leotta, M.; Clerissi, D.; Ricca, F.; Spadaro, C. Improving test suites maintainability with the page object pattern: An industrial case study. In Proceedings of the IEEE Sixth International Conference on Software Testing, Verification and Validation Workshops, Institute of Electrical and Electronics Engineers (IEEE), New York, NY, USA, 18–22 March 2013; pp. 108–113.
43. Parker, M.H.; Kepple, L.R.; Sklar, L.R.; Laroche, D.C. Automated GUI Interface Testing. U.S. Patent 5,781,720, 14 July 1998.
44. Yang, X.; Miao, Y.; Zhang, Y. Model-driven GUI Automation for Efficient Information Exchange between Heterogeneous Electronic Medical Record Systems. In *Information Systems Development*; Springer: New York, NY, USA, 2011; pp. 799–810.
45. Asatiani, A.; Penttinen, E. Turning robotic process automation into commercial success-case OpusCapita. *J. Inf. Technol. Teach. Cases* **2016**, *6*, 67–74. [CrossRef]
46. Kasper, M.; Correll, N.; Yeh, T. Abstracting perception and manipulation in end-user robot programming using Sikuli. In Proceedings of the IEEE International Conference on Technologies for Practical Robot Applications (TePRA), Woburn, MA, USA, 4 August 2014; pp. 1–6.
47. Polden, J.; Pan, Z.; Larkin, N.; Van Duin, S.; Norrish, J. Offline Programming for a Complex Welding System using DELMIA Automation. In *Robotic Welding, Intelligence and Automation*; Springer: Berlin/Heidelberg, Germany, 2011; pp. 341–349.
48. Huesman, J. *Spaceport Command and Control System User Interface Testing*; Technical Report for Launching Commercial and Government Owned Spacecraft: Fargo, ND, USA, April 2016.
49. Warner, P.C. Automatic Configuration of Programmable Logic Controller Emulators (No. AFIT-ENG-MS-15-M-024). Master's Thesis, Air Force Institute of Technology, Wright-Patterson Graduate School of Engineering, School of Engineering and Management, Wright-Patterson, OH, USA, 1 March 2015.
50. Corvin, C.M. A Feasibility Study on the Application of the ScriptGenE Framework as an Anomaly Detection System in In-dustrial Control Systems. Master's Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Tech-nology, Wright-Patterson, OH, USA, 17 September 2015.
51. Gallenstein, J.K. Integration of the Network and Application Layers of Automatically-Configured Programmable Logic Con-troller Honeypots (No. AFIT-ENG-MS-17-M-029). Master's Thesis, Department of Electrical and Computer Engineering, Air Force Institute of Technology, Wright-Patterson, OH, USA, 2017.

52. Girtz, K.; Mullins, B.; Rice, M.; Lopez, J. Practical application layer emulation in industrial control system honeypots. In *International Conference on Critical Infrastructure Protection*; Springer: Berlin/Heidelberg, Germany, 2016; pp. 83–98.
53. Wang, Z.; Wagner, A. Evaluating a Tactile Approach to Programming Scratch. In Proceedings of the 2019 ACM Southeast Conference, Kennesaw, GA, USA, 18–20 April 2019; pp. 226–229.
54. URSim. Available online: https://www.universal-robots.com/download/ (accessed on 25 November 2020).
55. RealVNC. Available online: https://www.realvnc.com/ (accessed on 25 November 2020).
56. jQuery Sortable. Available online: https://jqueryui.com/sortable/ (accessed on 25 November 2020).
57. Glumb robot simulator. Available online: https://github.com/glumb/robot-gui (accessed on 25 November 2020).
58. Corkill, D.D.; Gallagher, K.Q.; Murray, K. GBB: A Generic Blackboard Development System. *AAAI* **1986**, *86*, 1008–1014.
59. Yeh, T.; Chang, T.H.; Miller, R.C. Sikuli: Using GUI screenshots for search and automation. In Proceedings of the 22nd Annual ACM Symposium on User Interface Software and Technology, Victoria, BC, Canada, 4–7 October 2009; pp. 183–192.
60. Wilson, B.G. *Constructivist Learning Environments: Case Studies in Instructional Design*; Educational Technology: Enschede, The Netherlands, 1996.
61. Jaatinen, J.; Lindfors, E. Makerspaces for Pedagogical Innovation Processes: How Finnish Comprehensive Schools Create Space for Makers. *Des. Technol. Educ.* **2019**, *24*, n2.
62. Bradski, G.; Kaehler, A. OpenCV. *Dr. Dobb's J. Softw. Tools* **2000**, *25*, 3.
63. ABB Single-Arm Yumi. Available online: https://new.abb.com/products/robotics/industrial-robots/irb-14050-single-arm-yumi (accessed on 25 November 2020).
64. Brugali, D.; Scandurra, P. Component-based robotic engineering (Part I) [Tutorial]. *IEEE Robot. Autom. Mag.* **2009**, *16*, 84–96. [CrossRef]
65. Brugali, D. Model-driven software engineering in robotics: Models are designed to use the relevant things, thereby reducing the complexity and cost in the field of robotics. *IEEE Robot. Autom. Mag.* **2015**, *22*, 155–166. [CrossRef]
66. Sadik, A.R.; Goerick, C.; Muehlig, M. Modeling and Simulation of a Multi-Robot System Architecture. In Proceedings of the International Conference on Mechatronics, Robotics and Systems Engineering (MoRSE), Bali, Indonesia, 4–6 December 2019; pp. 8–14.
67. Aspridou, M. Extending BPMN for Modeling Manufacturing Processes. Master's Thesis, Business Information Systems, TU Eindhoven, Eindhoven University of Technology, Eindhoven, The Netherlands, 30 October 2017.