Creative Robotics

Robots for Skill Digitization

Von der Fakultät für Architektur

der Rheinisch-Westfälischen Technischen Hochschule Aachen

zur Erlangung des akademischen Grades eines
Doktors der Ingenieurwissenschaften

genehmigte Dissertation

vorgelegt von

Johannes Braumann

aus Salzburg, Österreich

Berichtende:
Univ.-Prof. Dr. Jakob Beetz
Univ.-Prof. Dr. techn. Sigrid Brell-Cokcan

Tag der mündlichen Prüfung: 15. Dezember 2025

Abstract

Digitization and automation are powering today's industrial fabrication. A core technology in this area is robotic arms - multi-functional machines that originated in the automotive industry and are now used in a wide range of fields.

Although robotic arms have been available as a technology for six decades, they have only recently found more widespread use among creative users. An essential factor enabling this development is the creation of new, domain-specific robot control interfaces optimized for the requirements of the creative industry.

Flow-based, visual programming platforms have brought parametric design and algorithmic approaches beyond large, high-tech offices to individual architects, designers, and craftspersons. The research presented in this thesis builds upon these technologies and integrates them with an environment for programming, simulating, and controlling industrial robots.

Generic robot programming software is optimized for industrial mass fabrication, which uses multiple robots to perform a series of comparably short, pre-programmed tasks with a complex, high-level control system keeping the production synchronized. In contrast, robotic processes in the creative industry often implement (mass) customization. They are highly geometry-informed, with the robot's movements being generated based on parametric geometry, generally resulting in complex, long-running production processes.

This complexity can only be managed through user-centric, accessible software that allows individuals with specific material or process knowledge to apply their expertise within the context of robotic fabrication. This process is referred to as "Skill Digitization" and is presented through a series of case studies.

A central challenge in achieving this goal is the closed system of robotic arms: As robot manufacturers do not publish their proprietary algorithms, it is necessary to reverse-engineer many technical aspects of robot control and to set up a custom software architecture that is

optimized to efficiently process, buffer, and visualize the data flows of robotic processes within the visual programming environment.

This thesis explores the potential of these specialized data flows within the context of the creative industries, with a focus on their viability, the benefits of accessibility, and the disruptive potential towards enabling entirely new products and services.

The developed software KUKA|prc serves as a proof of concept, implementing the approaches developed in the thesis and forming the foundation for the presented case studies.

Table of Contents

1 Introduction

This thesis presents a spectrum of work covering ten years, arrayed around the initial concept of bringing robot simulation and control to a flow-based visual programming environment. It is motivated by the author's effort to realize non-standard production processes within the scope of the creative industries on commercially available industrial robots.

The term "creative industry" is deliberately chosen to convey that the developed approaches are not limited to the initial scope of architecture and construction but extend to related fields such as industrial design, crafts, and trades, as well as entirely new areas incorporating individualized production that have yet to be defined.

The thesis is laid out as a reference to researchers and professionals within the creative industries. While Chapter 2 and the case studies in Chapter 5 are accessible to non-experts, Chapters 3 and 4 provide in-depth information aimed at readers with advanced knowledge in robotics and parametric design who are either looking to realize their tools for robot control or to customize/expand the author's software.

Complex information is generally conveyed visually and descriptively rather than through formulas and equations, while code is provided as actual snippets rather than pseudo-code.

The work's main contribution to this field of research is the development of complex data- and workflows through flow-based programming that resulted in the software KUKA|prc, which is used by numerous academic and commercial institutions today.

1.1 Research Questions

Robotic applications in industry are commonly programmed once and then repeated with little to no variations in the machine's movement, e.g., dynamic pick-and-place operations on a conveyor belt or spot-welding of a car chassis (Chapter 2.1). In contrast, the fabrication of architectural components involves a high amount of geometrical complexity (Chapter 2.3) – the possibility space of a parametric model is not limited to affine transformations but extends to the generation of

entirely individualized geometries, as done, e.g., by Züblin Timber (Chapter 2.4.3).

As such, the variations can no longer be realistically represented as numeric variables in the robot's machine code, instead requiring full-featured computational geometry libraries – not just for generating the robot's toolpath but also for generating the geometry itself in the first place.

During the initial phase of this development, the visual programming environment Grasshopper was starting to become an essential part of a designer's digital toolkit for dealing with complex geometries and automating repetitive tasks. While the basic process of using a visual programming environment to control a machine is not new (see Chapter 3.4), existing programs had a generic scope without emphasizing computational geometry. With no viable solutions available at the time, the initial, fundamental research questions were:

Is there merit in constraining the toolpath of a robot to parametric geometry through flow-based visual programming?

Initial experiments and workshops with custom scripts capable of generating robot code proved that there was significant interest not only from academia but also from commercial users (Chapter 3.2). These events also showed that a primary challenge in programming an industrial robot is the complexity of its CAD-CAM (Computer-Aided Design/Manufacturing) data workflow (Chapter 2.3), which led to the question:

Can programming and simulating a robotic arm be made accessible to the creative industry by implementing it within a visual programming environment?

Realizing such integration was primarily a technological task, as the entire robot code generation, inverse- and forward kinematics, etc., had to be reverse-engineered (Chapter 3.3). The data flows resulting from the flow-based visual programming paradigm led to a series of particular challenges towards facilitating immediate feedback and fluid interaction with the parametric model and its simulated robot (Chapter 4.3).

With this solid, algorithmic foundation, it was possible to concentrate on the user experience with a specific focus on the needs of the creative industry (Chapter 3.5), towards investigating new ways of interacting in real-time with a robot (Chapter 4.6) and implementing capabilities of new families of robotic arms such as collaborative robots (Chapter 4.7).

The final research question links back to the industrial concept of augmenting a task that a worker previously did with a robot but explores it within the context of the creative industry:

How can robotics advance and benefit the creative industries, resulting in entirely new approaches, products, and innovation?

This question is explored through a series of "Skill Digitization" case studies (Chapter 5) that fulfill the premise to different degrees in various disciplines.

As of now, the widespread application of robotics within the creative industry is still very recent, compared to the six decades of industrial robots in other industries, but is rapidly developing, as can be observed in the rising complexity of the author's projects and the integration of advanced features into "creative" software.

What sets the presented approaches apart from other research is the focus on the end-user within the visual programming environment: KUKA|prc is not simply a collection of useful robot-related functionality or a particularly efficient solution for a given problem but comes as a full-featured framework with a variety of implementations that make the software immediately useable for experimentation. Each observes the particularities of the respective host environments.

The primary contribution to the field lies, therefore, in the integration rather than in the individual features: Instead of exposing all features offered by an industrial robot, the software provides a core feature set that has been gradually expanded based on the feedback of the users and the requirements of the author's projects, making it natively suited to the demands of the creative industry.

Doing so is only possible by developing software not from a purely technical point of view but with a user-centered approach that is rooted in one's practice and experience.

1.2 Overview

Today, digitization allows even small enterprises to access a global market or to rent scalable, high-performance servers in the cloud, providing service with the same ease to dozens or thousands of concurrent users. Even many fabrication processes, from 3D printing to milling and circuit board manufacturing, are available globally on demand through service providers (Rudolph and Emmelmann 2017). This scalability greatly empowers startups, allowing them to access services without the up-front investment for servers or machines.

However, developing specialized machines that enable processes beyond the state-of-the-art requires teams of programmers, mechanical engineers, electrical engineers, and safety experts (Helbig et al. 2016) – exceeding the financial capabilities of many small enterprises and startups.

In automation, an alternative to special-purpose machines is industrial robots - generic machines equipped with various tools to cover a wide range of applications. The advantage of such an approach is that rather than having to control an entire machine, the user can focus on the machine's movement, its toolpath (Figure 1).

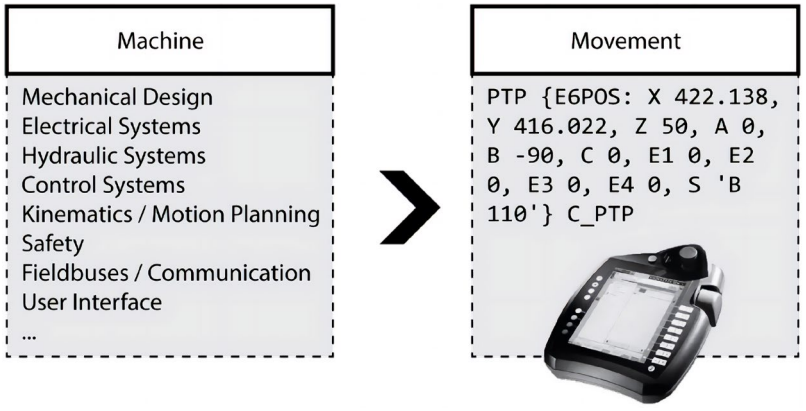

Figure 1: Industrial robots abstracting the complexity of building a custom machine to dealing with movement.

Industrial robots have been used by industry for more than six decades and have become commonplace in the automotive field and many other sectors. With increasing demand and competition (IFR 2018), these machines have become increasingly affordable, even for small enterprises, with the average unit costs falling by 11% within five years (Oxford Economics 2019). To program a simple robotic task, the user needs to move the robot to a given position, save it, and then repeat the process until a toolpath is defined – a process referred to as "online teaching."

While this strategy can cover generic industrial manipulation, there is an increasing interest in realizing new robotic applications from fields that have not used robots before. In the creative industry, architects and designers want to realize complex processes supporting mass customization, while craftspersons strive to apply their craft knowledge to machines.

This thesis has identified three existing families of robot control software to realize complex fabrication processes using robotic arms:

Application-specific software is provided only for existing industrial processes such as computer numerical control (CNC) milling and 3D printing. Frequently, existing Computer-Aided Manufacturing (CAM) software (Figure 2, top) is expanded to support six-axis robots in addition to simpler kinematics (Autodesk 2020c; Hypertherm 2020). This type of software is usually priced for companies that use it productively regularly, with a user experience that follows standards set by CAM software.

Generic robot programming software (Figure 2, bottom) is often sold by the robot manufacturers (KUKA 2020a; ABB 2020b) but may also be provided externally (OCTOPUZ Inc 2020; RoboDK Inc. 2020). Generic software allows the user to perform the teaching process (see Chapter 2.1.1) on a virtual robot in an accessible way, supporting it with functionality that, e.g., snaps the robot's tooltip to a geometry. However, programming custom toolpaths that go beyond the tracing of Computer-Aided Design (CAD) geometry requires the user to use the built-in scripting functionality. The complexity of programming, therefore, rises significantly once the user's requirements can no longer be covered by

(virtual) teaching. Similar to application-specific software, generic robot programming software is priced for industrial users.

Research-oriented software provides significant functionality and features that go beyond robotic fabrication toward trajectory optimization, real-time control, spatial mapping, etc. It is often available as open-source at no cost like ROS (Quigley et al. 2009), though commercial solutions like CoppeliaSim (Rohmer, Freese, and Singh 2013) also exist. Because of the number of features irrelevant to fabrication processes with a robotic arm, research-oriented software generally requires advanced knowledge of robotics and programming.

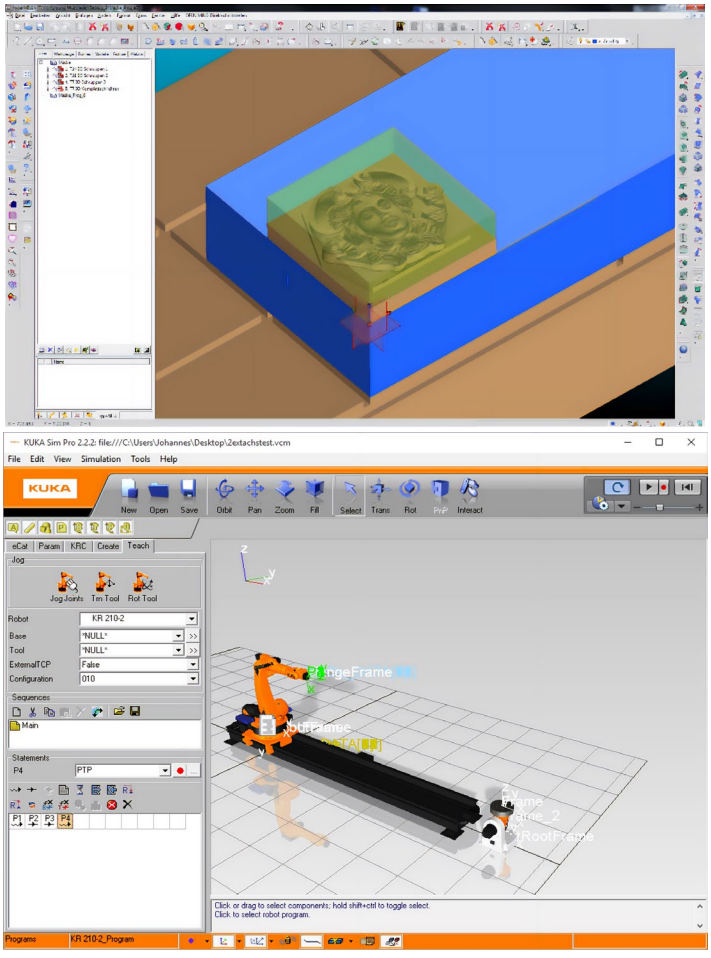

Figure 2: HyperMill as an example of CAM software (top), KUKA SimPro as an example of offline robot programming software (bottom).

For the creative industry, none of these approaches is ideal: While users can apply their knowledge of CAM software to application-specific software, its scope is limited, while research-oriented software requires advanced knowledge of robotics, inter-system communication, etc. On the other hand, generic software cannot easily represent mass customization and individualization without advanced programming know-how.

A comparable situation can be found in the area of computational geometry and geometry optimization in architecture. For a long time, the construction of geometrically complex buildings required consulting services by mathematicians and programmers, which were only affordable for high-end projects with large budgets.

Application-specific, architectural software like AutoCAD (Autodesk 2020b) did not support the envisioned complexity, while research-oriented software, in particular, custom scripting solutions such as Microstation VBA - e.g., for the Beijing Water Cube (Bull and Downing 2004) - exceeded the geometrical and programming know-how of architects. As such, a common strategy was using generic (CAD) software to construct and adjust geometry by manually interacting with its control points – e.g., for Kunsthaus Graz (Brell-Cokcan 2017) – making design iterations extremely time-consuming.

Visual programming environments became a catalyst that made advanced geometry accessible to smaller offices and individuals (Celani and Vaz 2012). Architects and designers were able to experiment intuitively with geometric optimization and discretization, thanks to the direct feedback provided by these programming environments. This process was supported by a community that assists new users and continually expands the scope of visual programming through new plugins and software.

This thesis originates from that creative community and documents the usability strategies, software architecture, domain-specific workflows, and feature integration of a robot simulation and control environment explicitly developed for visual programming (Figure 3) (Braumann and Brell-Cokcan 2014). Over ten years since the first related publication in 2009 (Brell-Cokcan et al. 2009), the resulting software KUKA|prc (Braumann and Brell-Cokcan 2011) has dramatically expanded in scope

and is used today by both creative as well as industrial users to prototype new robotic processes rapidly or to implement mass-customization (Braumann 2020).

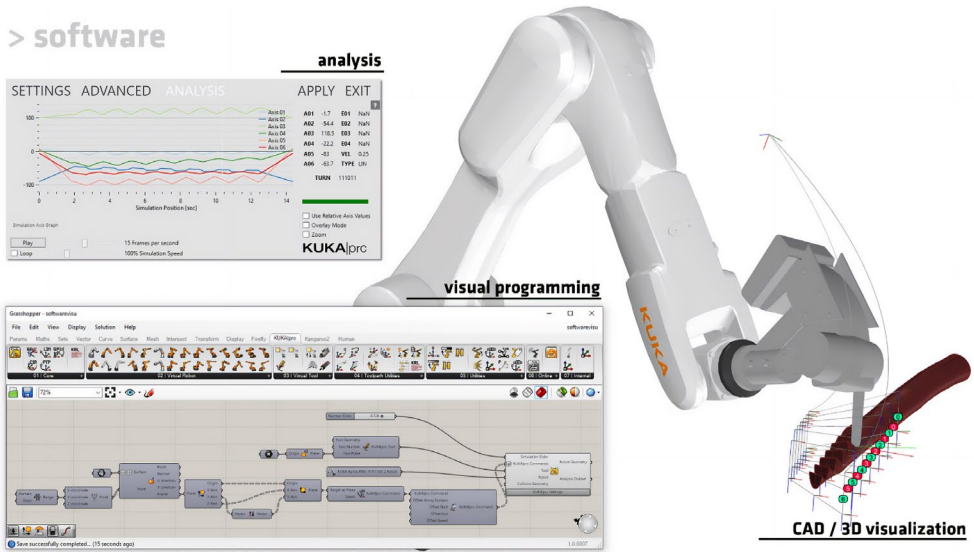

Figure 3: Visual programming of an industrial robot through KUKA|prc.

That development is closely linked to current advances toward "low-code" strategies (Fryling 2019) in a wide variety of fields that make technologies that were previously accessible exclusively to programmers available to a more extensive range of users.

1.3 Reading Guide

Chapter 2 provides an overview of robotic fabrication and the associated workflows in both industry as well as for industrial users. Chapter 3 focuses on visual programming and documents the development of KUKA|prc, from simple G-code generation to kinematic robot simulation. Furthermore, the user interaction of visual programming users with the robot simulation environment is explored. Chapter 4 provides a technical overview of the actual implementation of robot programming and simulation in visual programming, considering constraints such as the directed data flow. Finally, Chapter 5 presents four families of use cases that apply the potential of visual programming for robotic processes in different ways, followed by the conclusion and outlook in Chapter 6.

2 State-of-the-Art: Approaches towards Robotic Fabrication

In recent decades, robotic automation and digitization have focused chiefly on medium to large-scale enterprises and much less on small companies, the creative industry, and skilled crafts and trades (IFR 2018). As a result, workflows, processes, and pricing are aimed at larger companies' requirements and financial capabilities rather than small companies or even individual entrepreneurs.

This chapter provides an overview of three areas:

- Robotic fabrication in industry (Chapter 2.1)
- Tools for robot programming (Chapter 2.2)
- Computational design in the creative industry (Chapter 2.3)

These areas are then brought together in Chapter 2.4, enabling the CAD-based robot programming approaches used today in architecture and construction.

2.1 Robotic Fabrication in Industry

Robotics is a comprehensive and interdisciplinary field of research that is constantly developing, e.g., the easy availability of drones enabling entirely new research trajectories into swarms of machines (Waibel, Keays, and Augugliaro 2017). In comparison, the sub-field of robotic fabrication is evolving at a slower pace, as its primary customers are established, large companies like the automotive sector, who value reliability above the cutting edge of innovation. Robotic processes often only assume auxiliary roles, such as loading other CNC machines.

While startups are pushing into the market with innovative robotic concepts, especially in the area of collaborative robotics, like the Franka Emika Panda with new UI concepts (Schmidbauer, Komenda, and Schlund 2020), it is challenging to move beyond academic users towards large-scale industry, as per an analysis of Rethink Robotics' bankruptcy in 2018 (Juetten 2018). Currently, 57% of the industrial robots market is

held by just four companies: Fanuc, ABB, KUKA, and Yaskawa (Stiehler and Gantori 2020).

Robotic fabrication is mainly associated with robotic arms, whose origins are primarily shared with other CNC machines. In 1952, on behalf of the US Army, the MIT Servomechanisms Laboratory converted a commercially available mill into a machine with simultaneous three-axis movements (Parsons and Stulen 1958). Only two years later, George C. Devol patented a "general purpose machine that has universal application to a vast diversity of applications where cyclic control is to be desired" (Devol 1961), which resulted in the Unimate robot (Figure 4, left). It was first sold in 1961 to General Motors.

In 1969, the Stanford Arm was already presented with six degrees of freedom, driven by DC motors (Feldman et al. 1969). Continuous path motion was introduced in 1973 by Swedish company ASEA's IRB-6 robot, thus enabling continuous processes like arc welding and milling.

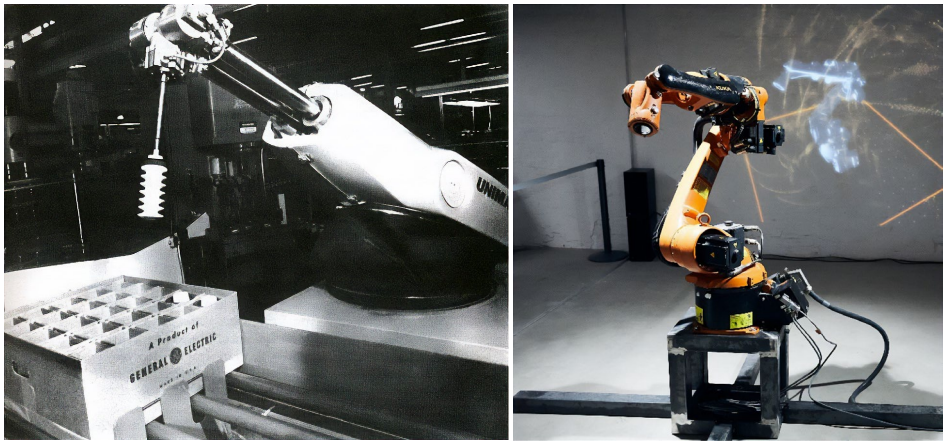

Figure 4: Unimate 5-axis industrial robot, 1961 (left, Unimation). KUKA KR5 arc HW 6-axis industrial robot, 2005 (right).

The concept of the industrial robot has since been further iterated based on user requirements and towards enabling new ways for fabrication and assembly, particularly for the automotive industry, which – with 125,700 robots per year (IFR 2018) – is still the primary user of robotic arms. Looking at the example of a KUKA industrial robot (Figure 4, right), such as the KUKA Quantec-2 series, a robot consists of at least three components: The robotic arm itself, the control cabinet, and the teach

pendant, through which the user interacts with the machine. A Quantec-2 robot consists of six axes that can be moved at a maximum of 115-260 degrees/second with a repeatability of ± 0.05 mm as per ISO 9283 (ISO 1998; KUKA 2019).

In this case, the KUKA KRC4 controller using the KUKA KSS8.6 software base contains the motor drivers and an industrial PC running Windows 10 and VxWorks, which handles all time-critical tasks in hard real-time. User interaction is done mainly through the teach-pendant, the KUKA smartPad-2, which acts as a thin client that connects to the controller via a wired Remote Desktop connection. In addition to various buttons, it offers a 6D mouse for intuitively moving the robot in 3D space and a safety-relevant E-Stop button. A typical application for such a robot would be to perform spot-welding as part of an automotive fabrication line, with pre-programmed movements that are called by the remote PLC managing the production line.

Within the context of CNC machines, robotic arms are neither the fastest nor most accurate machines but are used for their flexibility and affordability (Iglesias, Sebastián, and Ares 2015), as it is possible to equip them with a wide range of tools and program them in languages that allow more complexity than simple DIN 66025 G-code (DIN 1988).

2.1.1 Exemplary Workflow: Online Robot Programming

"Online Programming", also referred to as "Teaching", is the most common way of programming a robot in industry. The programming process takes place entirely on the robot's teach pendant, where the user can save the robot's current position in a file and then repeat that process to generate a toolpath (Pan et al. 2012). Additional logic like input-output operations, loops, and conditional expressions can be added through the user interface or by entering the code directly as text in the robot's programming language. The advantage of teaching is that the positions are taken from the live object, avoiding the difficult task of matching a CAD model with the actual physical environment. This approach also benefits directly from the fact that a robot's repeatability is significantly better than its absolute positioning in 3D space: A saved position can be approached again by the robot within very low

tolerances (repeatability), while the robot's ability to precisely move to a given coordinate in space is comparably limited (absolute accuracy). The disadvantage of such a process is that each position has to be recorded individually. New technologies like the robot-mounted 6D mouse ready2Pilot (KUKA 2020b) facilitate moving the robot to the desired position. However, subtractive processes or processes requiring many hundreds or thousands of points are tedious or impossible to program.

2.1.2 Exemplary Workflow: Offline Robot Programming

Offline programming in environments like KUKA.Sim translate the process of teaching from a physical into a virtual environment. This allows the user to quickly create toolpaths by directly snapping to points in a CAD model or by tracing curves or edges of geometry with the robot's tool. Plugins like KUKA.CAMRob (2009) or ABB's Machining PowerPac (2020a) also allow the import of standardized G-code (DIN 66025, APT-CL, and others) and its translation into robot-specific code. However, the core challenge of a common offline programming approach is that a very accurate 3D model of the environment is needed to ensure an accurate simulation and that the software is optimized for hands-on, individual programming rather than batch processing, as robots in industry are generally used for mass-fabrication where the programming process only needs to take place once.

2.2 Robot Programming Tools and Middleware

There are numerous approaches towards robot control, each with a particular focus. Examples are the Open Robot Control architecture (Nilsson and Johansson 1999), which covers not just the robot's programming via an offline interface but also the motor control, building upon MATLAB and Simulink. Mitsi et al. (2005) present an offline programming workflow for welding that builds upon a combination of SolidWorks as the CAD environment, path-planning in Visual Basic, and an inverse kinematics solver written in Fortran. V-REP (Rohmer, Freese, and Singh 2013), now CopeliaSim, uses a highly modular approach that attaches Lua scripts to individual scene objects – which may be robots, but also (simulated) sensors or shapes – thus defining their behavior, as

well as providing a connection to ROS. ROBOLAB (Kucuk and Bingul 2010) and the Sunrise Toolbox (Safeea and Neto 2019) extend MATLAB with the capability of simulating industrial robots and cobots (collaborative robots), while JavaKUKA (Hua 2019) provides a robot simulation and code generation library for Java. Beyond these selected environments and architectures, a significant number of middlewares facilitate the development of new robotic machines and processes (Elkady and Sobh 2012), the most popular of them being ROS (Quigley et al. 2009). While, e.g., robotics applications built on MATLAB focus on mathematics and signal processing, ROS fosters easy interconnectivity between sensors and actors.

Due to the wealth of robotic applications and programming environments, there is no single benchmark architecture but separate solutions that greatly vary in scope – from library to middleware to commercial software – and each serves a domain-specific need.

2.3 Computational Design for Fabrication

While architects and designers have dreamed of robotic fabrication for more than 100 years, as evidenced by the work of Villemard (1910) (Figure 5), the broader proliferation of robotic arms in the creative industry has only begun in the late 2000s and early 2010s with the research at ETH Zurich (Bonwetsch et al. 2006), at a time where robotic arms were already very mature, industry-proven machines.

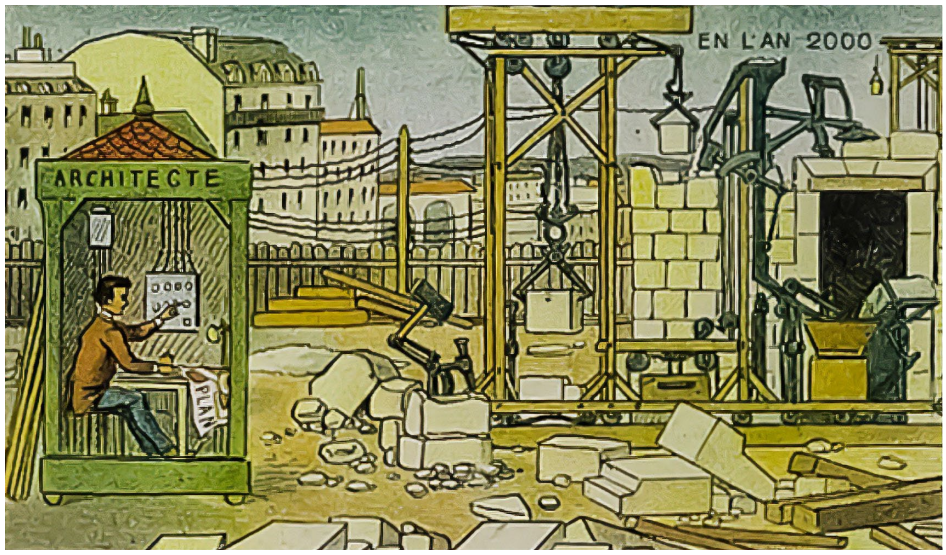

Figure 5: Villemard's vision of an architect controlling the construction site through robotic labor in the fictional year 2000.

However, while robots in industry are mainly applied for mass fabrication, the creative industry has a great need for and interest in mass customization and individualization, which is often not covered by standardized industrial workflows. As such, the state of the art in robotic fabrication is tightly linked to the evolution of generic computational design tools rather than specific fabrication software or hardware developments.

While drafting software like Autodesk AutoCAD helped automate and digitize the drafting process, 3D software like 3ds Max (2020a), Cinema 4D (2020a), and later Rhinoceros 3D (2020a) enabled architects in the 1990s to design highly complex 3D geometries. However, until the mid-2000s, dealing with the fabrication of complex, individualized geometries often required the consultancy of specialized offices that employed mathematicians and programmers to solve the architects' challenges or alternatively limited such technologies to large offices that had access to specialized in-house staff (Glymph et al. 2004).

A particularly relevant development during that time was the use of high-end engineering software CATIA (2020) by Gehry Technologies for solving the architectural challenges of Gehry's highly complex designs, e.g., the developable metal strips of the Guggenheim Bilbao (Glaeser and

Gruber 2007) built-in 1997. Due to the high investment in these technologies, Gehry Technologies also started offering their services to external offices like HdM, SOM, and others (Chang 2015) and published their CATIA-based software tools as Digital Project in 2004 (Figure 6, left). While CATIA/Digital Project could automate digital fabrication workflows, its high cost was prohibitive to many smaller offices (Day 2004).

Around the same time, David Rutten published his RhinoScript manual (Rutten 2004), presenting various examples of how to use VBScript within Rhinoceros 3D from the point of view of a creative user. This opened up scripting as a new tool for architects and was used to design and fabricate many buildings. Digital artists like Marc Fornes relied on RhinoScript (Fornes 2006) to develop structures consisting of many individual modules, often laser-cut out of sheet materials and intelligently assembled, inspiring both architects and designers.

In parallel, the early 2000s brought the beginnings of the practical application of BIM (Building Information Modelling), more than 25 years after the process was first coined as a "Building Description System" by Eastman et al. (1974), through software such as Charles River Software's Revit (later acquired by Autodesk). It offers an "object-oriented" approach to drafting, with the designer drawing actual objects like walls, containing metadata relating to costs, materials, etc., and allowing the change of parameters such as wall thickness (Chuck Eastman et al. 2008). However, BIM software initially focused on fields like planning, structural/HVAC engineering, and lifecycle management and has only since grown to support the actual digital fabrication of the BIM model (J. Lee et al. 2019).

In particular, within the context of this thesis, the most crucial development of the early 2000s in the field of computational design was the release of two visual programming environments that were made explicitly for architecture and design, Generative Components (2003; Aish and Woodbury 2005) (Figure 6, right), integrated into Bentley Microstation, and Grasshopper (2020) – released 2007 as Explicit History – integrated into McNeel Rhinoceros 3D. Both environments became highly popular, opening up algorithmic and generative design to users without specific programming knowledge.

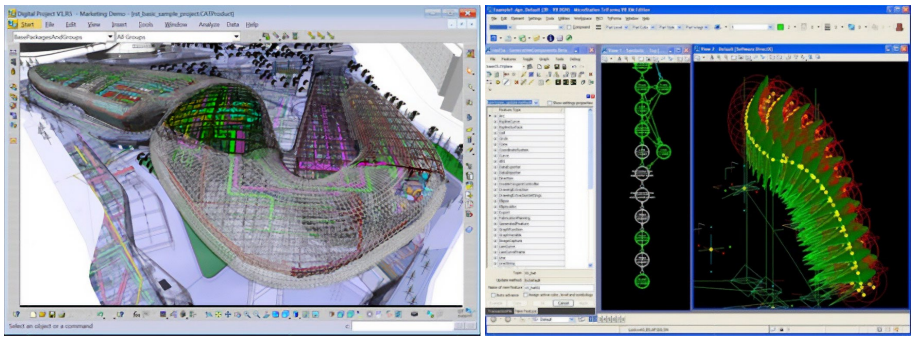

Figure 6: Generative design for architects: Gehry Technologies Digital Project BIM software (left), Bentley Generative Components visual programming (right).

While Generative Components paved the way for generative design in many high-end architectural offices, ultimately, Grasshopper had a more profound impact on the community and, 13 years later, is considered a standard tool for designers and architects around the world, whereas Generative Components is hardly visible anymore. The reason for that might be the significantly lower cost of its base CAD software, Rhinoceros 3D, which led to a larger community that actively kept expanding the range of Grasshopper into areas such as robotic fabrication.

2.4 Towards Robotic Fabrication

While research into robotic fabrication in architecture has been ongoing since the 1980s (Bock 1988), the results did not reach the larger community of the creative industry but were primarily rooted in academic research and engineering applications. In addition to architecture, there have been highly visible robotic installations, such as the 1999 fashion show by Alexander McQueen (Mower 2018) which involved a model and two spray-painting robots. These installations have reached a broad public, but have remained singular events that have remained inaccessible to the larger community.

Starting in the mid-2000s, the chair of Fabio Gramazio and Matthias Kohler at ETH Zurich became widely known for their robotically stacked brick walls that went beyond regular brick bonds, realizing irregular,

individual parametric patterns through robotic fabrication (Bonwetsch, Gramazio, and Kohler 2007; Bonwetsch et al. 2006). Shortly after that, more architectural faculties started to engage in research into robotic fabrication, such as Harvard, MIT, University of Michigan, ICD Stuttgart, and TU Vienna.

This effort was primarily of an academic nature, with engineering-focused literature of that time mentioning neither robotic arms nor the more universal term of industrial robots within the scope of automation in construction (Bock 2008).

The work of this group of researchers, originating from within the creative industry, resulted in many plugins and scripts that started making robotic fabrication accessible through an easy-to-use visual programming environment in the early 2010s, like the author's KUKA|prc (Braumann and Brell-Cokcan 2011), HAL (Schwartz 2013), and supermatterTools (Pigram and McGee 2011). While supermatter Tools was primarily used by its developers for various projects at the University of Michigan and the University of Technology Sydney, KUKA|prc and HAL were developed to be utilized by "external" non-expert users. A primary difference between these tools is that HAL aims to support many robot vendors through a single programming system, while KUKA|prc has been built exclusively for KUKA robots.

Compared to the programs presented in Chapter 2.2, these software tools differ from existing solutions as they provide the opportunity to rapidly create, simulate, and evaluate non-standard robotic fabrication processes based on parametric geometry through an accessible programming environment.

In 2011, Sigrid Brell-Cokcan and the author founded the Association for Robots in Architecture as a platform with the goal of making robots accessible to the creative industry (Brell-Cokcan and Braumann 2018). That effort has since led to the establishment of ROB|ARCH, a biannual conference series that was first held in 2012, the accompanying books published by Springer (Brell-Cokcan and Braumann 2013b), and the continued development of the parametric robot control software KUKA|prc.

Today, the Association for Robots in Architecture has over 150 member institutions (Figure 7), and KUKA|prc is used over 1500 times per day – by students, researchers, and professionals. Robotic fabrication has become part of the curriculum at many architectural faculties, and specific Master's programs have been set up at IaaC, ICD Stuttgart, RWTH Aachen, and others.

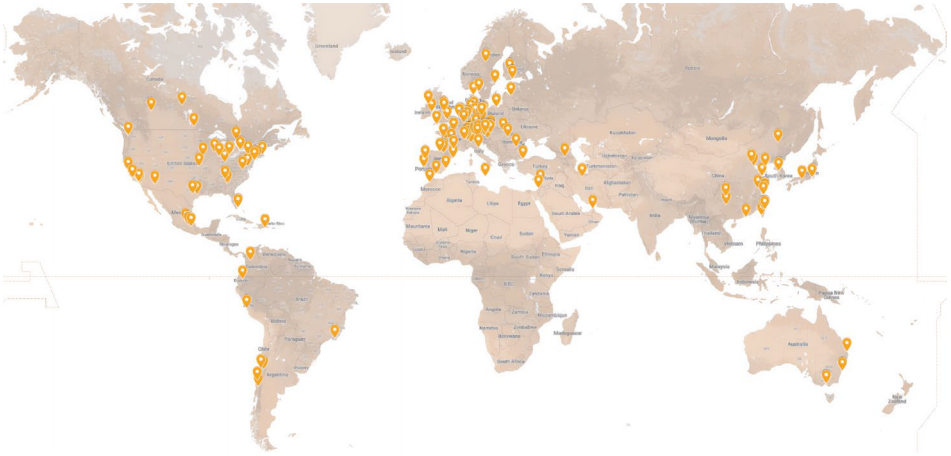

Figure 7: Map of registered institutions using KUKA|prc worldwide.

However, there is still decidedly less activity in the creative industry beyond architecture and, therefore, an even more significant innovation potential. The particular challenge is fragmentation: Where architecture is a relatively large and homogenous field, other communities within the creative industry are much smaller with more diverse requirements.

In architecture, CAD-CAM workflows are the standard within the scope of digital fabrication in architecture, working from a CAD model towards manufacturing. Visual programming has now started to supplement and even replace that process.

2.4.1 Exemplary Workflow: CAD-CAM

Traditional CAD-CAM workflows are commonly used in architecture, e.g., large-scale additive or subtractive manufacturing using a robotic arm. In general, a CAD-CAM workflow consists of three software components. A CAD environment within which the geometry is generated, either through a manual modeling process or a generative

algorithm. The design is then exported to a CAM environment, which offers a series of fabrication strategies, e.g., roughening passes to rapidly remove material with a cylindrical milling tool and finishing passes to create a smooth surface finish with a ball-head milling tool in the case of milling. Some passes can be assigned automatically (e.g., drilling processes to cylindrical holes), but a manual assignment is generally necessary.

Another core parameter is the 3D definition of the stock model (i.e., the original block of material) and process parameters like speed and step-down, which profoundly impact the quality, safety, and speed of a process. This results in a 5-axis toolpath, representing each position with a point, signifying the tooltip, and a vector, signifying the tool axis. Geometrically, a 5-axis machine has only one possible kinematic configuration to reach a position defined that way. However, robots with six or more axes provide an unlimited number of kinematic configurations.

Therefore, an additional software component is the robot simulator, where robot-specific parameters such as the chosen kinematic configuration can be set and simulated, and ultimately, machine code for controlling the robot is output. These three software components form a unidirectional workflow from design to robotic fabrication. Such a workflow works well for individual objects that are either of high value or fabricated in high numbers. However, it causes challenges when processing many elements, as human error can lead to collisions and invalid programs.

Additionally, the separation into 2-3 modules (see also Figure 9) is challenging to novice users, as they need to create geometry, assign toolpath strategies, set robot parameters, and only then see whether the robot is even capable of fabricating a piece, as the 5-axis toolpath does not necessarily guarantee that the 6+ axis robotic arm can reach a given position without collisions.

2.4.2 Exemplary Workflow: Flow-Based Visual Programming

Most common visual programming environments used in the creative industry, such as McNeel Grasshopper and Autodesk Dynamo, create a

directed, acyclic graph. The underlying programming paradigm of flow-based programming was first developed by J.P. Morrison (2010) in the late 60s for IBM, building upon the concept of co-routines that was developed even earlier (Conway 1963).

In Grasshopper's implementation of flow-based programming, operations are represented as function blocks with inputs on the left and outputs on the right side. When an input changes, the downstream components refresh automatically to reflect the change in input values. This creates a highly reactive system that lends itself to a very intuitive interaction with data and geometry. Commonly, such function blocks include geometric operations like, e.g., creating a point out of three numeric values representing its XYZ coordinates, but through plugins like the author's KUKA|prc, the range of function blocks can be expanded to include robotic fabrication, thus defining, e.g., the robot's programmed position through a local coordinate system.

The advantage of such an approach is two-fold:

Geometry and toolpath strategies can be generated simultaneously so that it is possible to have the geometry respect specific fabrication-related parameters intelligently. In contrast, the fabrication-related data can be automatically assigned to the relevant geometric features. This automatically translates both into the potential to be used for the automated batch-generation of robot control data files and provides immediate feedback on the feasibility of the developed fabrication process.

As the user is defining their fabrication strategy, the potential scope of robotic fabrication is expanded beyond the functionality offered by CAM software, e.g., milling, wire-erosion, and plasma cutting. However, this also leads to increased complexity for the user, as developing such strategies requires in-depth robotic knowledge, material expertise, and proficiency with the visual programming environment.

2.4.3 Synergies and Challenges

Over the past decades, industry has built significant know-how in using robotic arms and their integration into larger fabrication environments,

such as in the automotive industry. While individualization is used in industry for various applications (Da Silveira, Borenstein, and Fogliatto 2001), these developments are mostly built on custom, purpose-built software rather than accessible programming environments. There are commercial software packages for various tasks like milling, 3D printing, 3D scanning, etc., but they only cover the most common, commercially relevant robotic tasks.

In comparison, the creative industry has used the newly developed tools for programming robots through visual programming environments to apply its knowledge in geometry, individualization, and generative design to robotic processes.

KUKA|prc users who are successful commercial examples for that development are, amongst others:

- Branch Technology in the US, developing highly efficient, rapid 3D printing strategies that build up a lattice structure instead of a layered strategy (Figure 8, bottom).
- CarbonAxis in France, creating an innovative process for highly flexible carbon fiber layup.
- Mobbot in Switzerland, developing large-scale 3D printing processes for concrete.
- Züblin Timber, adapting their custom robotic fabrication pipeline for parametric production.
- Aeditive, using the software to let robots produce customizable shot-crete parts (Figure 8, top).

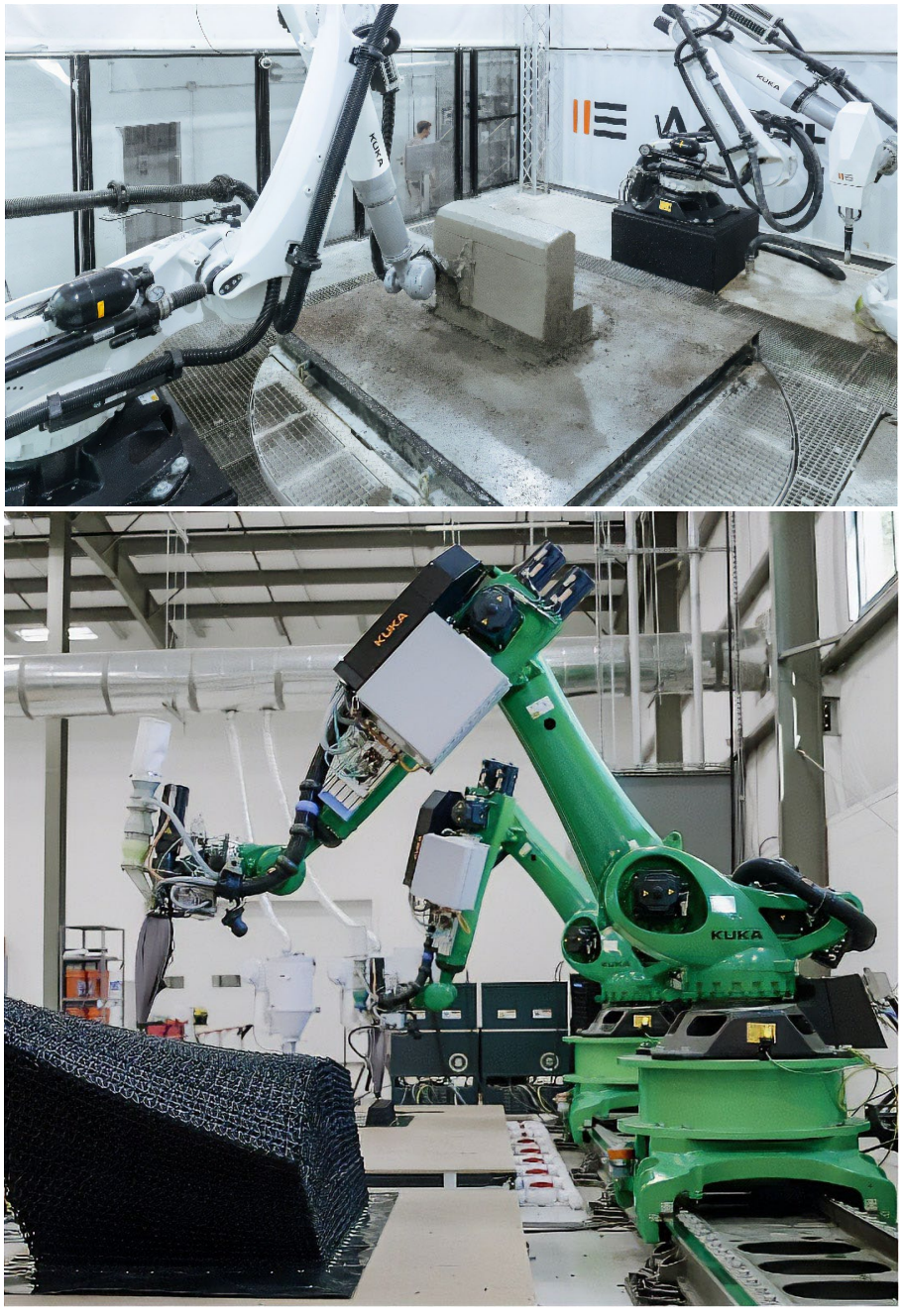

Figure 8: KUKA|prc used for customized shot-crete construction elements at Aeditive (top, Aeditive) and large-scale 3D printing at Branch Technology (bottom, Branch Technology).

This process has not gone unnoticed by the industry, so large companies like Boeing and Adidas are becoming users of KUKA|prc and actively looking for researchers with both creative and technical backgrounds.

While previously there has been a rather strict separation between designers and fabrication engineers, the deployment of such tools now creates a more even, collaborative environment that also fosters innovation by encouraging designers to implement fabrication-related parameters into their designs, thus greatly facilitating fabrication (Braumann and Brell-Cokcan 2015).

This development can be classified as part of the trend towards "low-code" (Fryling 2019), towards bringing new, digital technologies to a broader range of users while also allowing existing industries to prototype and develop at a fast pace.

The specific challenges towards enabling low-code robotic fabrication are found in the software architecture and dataflows necessary to represent robotic processes within a flow-based programming environment.

3 Flow-Based Programming for Parametric Robot Control

This chapter is divided into five main sections. Chapter 3.1 outlines the author's motivation towards developing KUKA|prc based on his experience with industry-standard robot programming software.

Based on that, Chapter 3.2 presents the evolution of the developed system for parametric robot control in four steps, from machine-agnostic toolpath design to an accurate robot simulation:

- Geometric toolpath design, post-processed by the CAM software
- G-code generation, post-processed by a robot simulation environment
- KRL generation
- Kinematic robot simulation

Chapter 3.3 provides a high-level overview of the mathematics behind the robot's kinematics. Finally, Chapter 3.4 explores the particularities of user interaction within the flow-based visual programming environment Grasshopper, with Chapter 3.5 elaborating on the specific implementation of KUKA|prc within that environment.

3.1 Motivation

The development of KUKA|prc was strongly influenced by the impact that visual programming had on the field of architecture towards the end of the 2000s and the complications that arose from the use of conventional CAD-CAM software for robotic fabrications. At that time, TU Vienna's Faculty of Architecture, with Sigrid Brell-Cokcan as the project leader and the author as a graduate researcher, received funding for the "Five-Axis Modelmaking" research project. Thus, the university acquired a KUKA KR60-HA robotic arm with a milling spindle, turntable, and automated tool-changer.

On the software side, TU Vienna became the first user of KUKA.CAMRob in Austria, a plugin for the robot simulation software KUKA Sim Pro that allows the user to import five-axis toolpaths, adjust the reachability strategy, and then output it in robot code. CAMRob also came with a

software component on the robot's side that implemented a ring buffer that would be continuously fed from the KUKA Sim Pro-generated BIN or CSV file containing the robot's positions without having to load the entire toolpath into the robot's limited memory, thus allowing highly complex toolpaths consisting of hundreds of thousands of positions (KUKA 2004). As the CAM solution, TU Vienna acquired licenses for OpenMind HyperMill (2020), a high-end, multi-axis milling software. This resulted in a software workflow representing the state-of-the-art robotic CAD-CAM process and was later encountered at Bamberger Natursteinwerke (see Chapter 5.2), where robots were used to mill large-scale stone sculptures.

However, for the purposes of an academic institution, the workflow (Figure 9) was not ideal as it required the project team to teach highly specific robot and CAM software to architecture students with no previous knowledge of robotics or CAM (Brell-Cokcan and Braumann 2013a). As such, a large part of the semester was spent on teaching just the handling of the software, limiting the output to only a few realized projects towards the end of the term. Additionally, the software was only available within a specific computer lab that was not accessible to students outside of the lecture hours and provided only a limited number of software licenses, which became apparent with the increasing popularity of the robot courses. As the scope of students' projects went beyond simple milling, the students had to creatively circumvent the limitations of the CAM software by creating 5-axis toolpaths as ruled surfaces in Rhinoceros 3D and then referencing these surfaces as swarf cuts in HyperMill. This demonstrated a specific need for creating interfaces that would allow the students to develop their particular fabrication strategies and link them with additional simulation tools (Brell-Cokcan and Braumann 2013a) without being limited to specific CAM cycles.

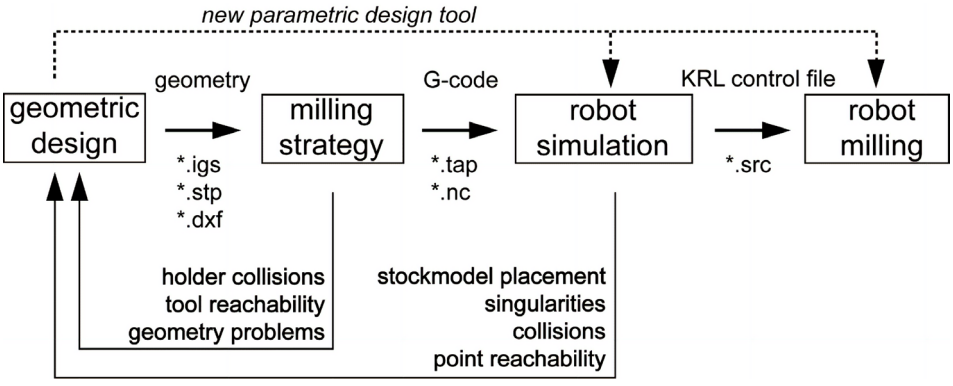

Figure 9: CAD-CAM workflow for robotic CNC processes.

3.2 Evolution

This section presents the chronological evolution of the developed software tool from the initial representation of toolpaths through ruled surfaces to the native generation of code in KUKA Robot Language.

3.2.1 Geometry-Based Toolpath Design

The initial approach towards enabling more control than CAM software usually provides was to intelligently create geometry that represents specific multi-axis toolpaths, thus being able to create toolpaths directly out of parametric geometry. This is built upon HyperMill's 5-axis swarf milling strategy. Swarf milling also referred to as flank milling (Waldt 2005; Bedi, Mann, and Menzel 2003), is a process primarily meant for the finishing of surfaces by moving a cylindrical tool along the rulings of a ruled surface. Thereby, the material is not removed layer by layer but in one cut – optionally with several step downs. In HyperMill, a swarf milling process is generally applied to a ruled surface, with additional parameters setting the number of step-downs and whether the tool's axis or flank moves along the rulings of the surface or normal to it.

Swarf milling provides an interface that can be used to define arbitrary multi-axis toolpaths easily. Generally, this was done by defining two curves: One curve that would represent the position of the tooltip and another curve that would define an arbitrary point along the tool's axis.

By lofting the curves together, a ruled surface is defined that may be used as the input for a swarf milling process.

The StackIt project (Brell-Cokcan and Braumann 2010) was developed using Grasshopper to create a stackable structure via swarf milling out of a block of XPS foam (Figure 10). In Grasshopper, the parametric ruled surfaces were evaluated and optimized for aesthetic criteria and production-related aspects, such as collisions between the tool holder and the stock model. They were then exported into HyperMill to generate the robot's milling code.

The added value of using Grasshopper for StackIt is that while the user directly designs the toolpaths, extra code visualizes the entire structure so that it becomes immediately visible how a local parameter change affects the global geometry. In parallel, the resulting toolpaths are evaluated regarding parameters such as the maximum depth of each cut. This value can then be used to optimize the toolpaths to ensure the structure can be fabricated using an available milling tool.

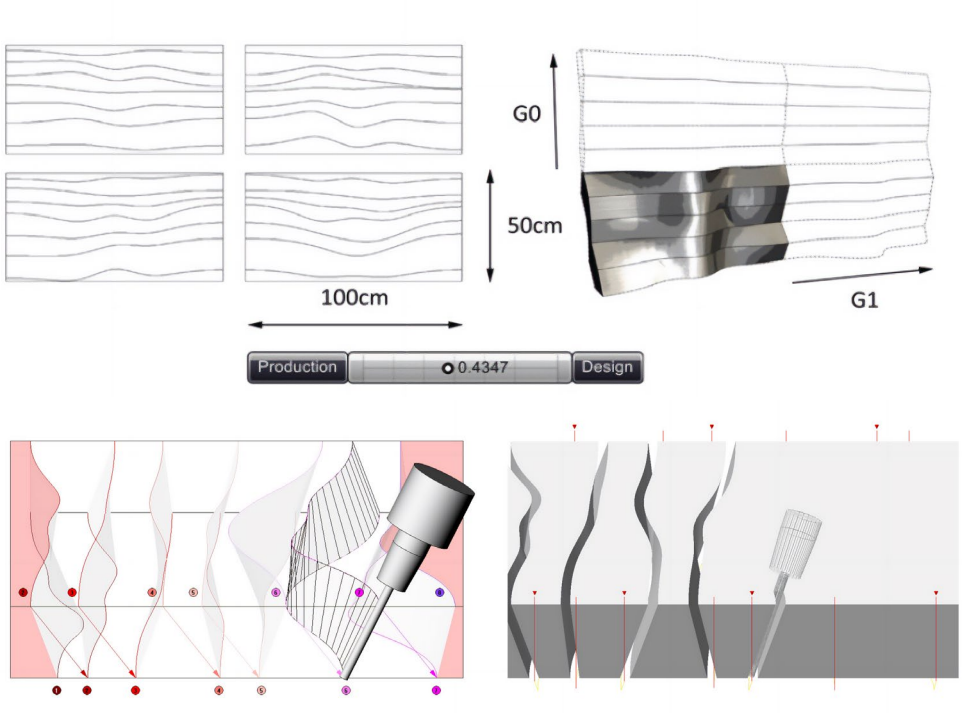

Figure 10: StackIt prototype based on parametric ruled surfaces.

3.2.2 G-Code-Based Toolpath Design

For geometry-based toolpath design, the CAM software only acts as an expensive postprocessor, translating geometric data into the five-axis G-code required by the robot simulation environment. CAM software still provides the added value of being able to simulate the material removal; however, for many processes, such a simulation is not necessary. To circumvent the CAM software directly, the G-code (Code 1) - saved as a human-readable file format with TAP extension – was analyzed with the goal of reverse engineering its structure. The official KUKA product documentation states that while the DIN 66025 is observed, not all commands are integrated (KUKA 2004).

```
1 N6 G0 X-20.044 Y183.704 Z59.029 I0.017 J-0.097
K0.995
2 N7 X-20.044 Y183.704 Z9.029 I0.017 J-0.097 K0.995
F2000
3 N8 X-18.083 Y173.898 Z8.986 I0.017 J-0.097 K0.995
```

Code 1: Syntax of movement commands in G-code.

The `N` value denotes the line number and increments with each line. `G0` at the beginning represents a rapid movement, with the actual speed set by the KUKA CAM.Rob process parameters. A regular, linear movement is assumed if `G1` or no G-value is provided. For any movement, `XYZ` represents the Cartesian target position in the current base coordinate system, and `IJK` the tool-axis as a unitized vector where I equals the vector's X value, J Y, and K Z. Finally, the F value changes the current speed to the subsequent value in mm/min.

Assembling that logic in Grasshopper is quickly done by formatting a `string` accordingly.

To facilitate the use of the G-code generating script, two toolpath strategies were created: A swarf-milling code (Figure 11) that would accept a pair of curves as the geometric input and numeric values for the spindle speed, feed rate, curve resolution, and tangential extension at the start and the end of the toolpath, and a surface milling code that takes an untrimmed NURBS surface as the input and would follow its

isocurves in a zig-zag pattern, deriving the tool orientation from the normal vector at each position on the surface. In addition to the numeric parameters of the swarf-milling, surface milling also requires a value for the number of isocurves to trace.

This created an accessible tool allowing users to circumvent HyperMill for non-standard applications. However, the code still required postprocessing through the KUKA SimPro/CAMRob environment.

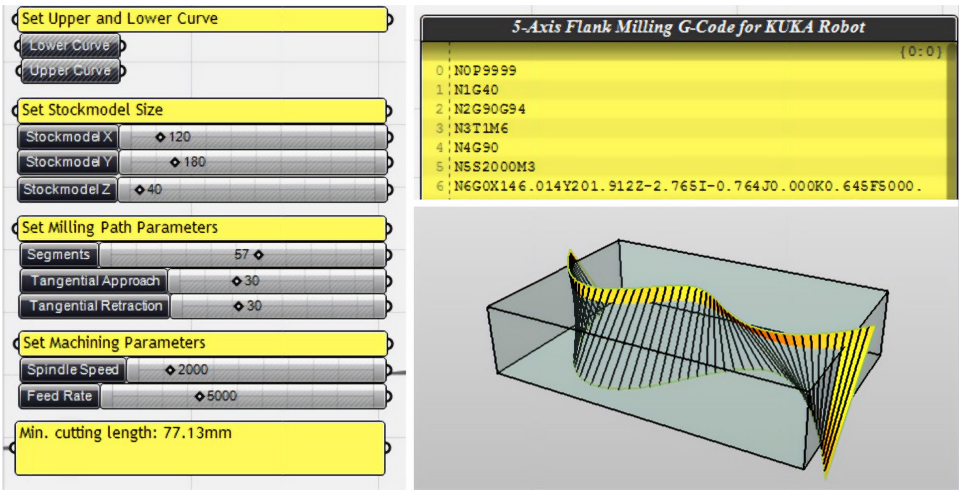

Figure 11: Grasshopper code for 5-axis flank milling, presented at the 4th Milling Conference in Vienna, Austria

3.2.3 KUKA Robot Language Generation

The direct generation of KUKA Robot Language (KRL) (Brell-Cokcan and Braumann 2010) finally makes the process independent of external tools, as code can be directly executed at the robot.

KRL is a proprietary programming language for KUKA robots, whose structure is similar to Pascal (Mühe et al. 2010). Unlike G-code, which is primarily a sequential order of different kinds of operations (motion, IO, etc.) and the possibility to “goto” a specific line number, KRL supports more complex code containing loops, conditional expressions, interrupts, variables, and subprograms.

While that functionality is essential for conventional robot programming, it is less so when a visual programming environment is used because the entire process logic is represented as components, resulting in a relatively simple sequence consisting primarily of motion commands. As such, it is unnecessary to analyze and understand the entire programming language; instead, focus on the definition of robotic motion.

By default, there are two ways to define a robot position in KRL: as a Cartesian position or as a collection of axis values. Cartesian positions are defined through the data structure `E6POS`. Every `E6POS` contains floating-point values for X, Y, Z, A, B, C, E1, E2, E3, E4, E5, E6, and integers for S and T.

XYZABC consists of the Cartesian coordinates XYZ and the three angles ABC, signifying Euler angles following the roll-pitch-yaw convention, i.e., rotating around the coordinate axes. Together, these six values define a coordinate system, also referred to as a frame, which uniquely defines the position of a robot tool in 3D space. However, the robot's posture is not automatically defined by it, so some ambiguousness remains even with a fully defined tool position and orientation (see Chapter 3.3).

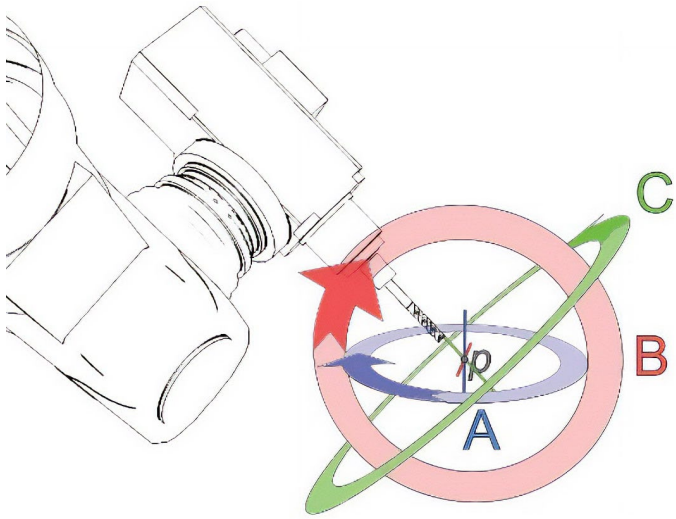

Figure 12: XYZABC values defining the position and orientation of a robot's tool center point.

The second relevant data structure is E6AXIS, containing floating-point values for A1, A2, A3, A4, A5, A6, E1, E2, E3, E4, E5, E6. It defines the exact axis value in degrees for A1-A6, thus uniquely defining the robot's position.

Within a CAD-CAM workflow, the E6AXIS structure is rarely used beyond the initial start position. Cartesian motions are used for all other positions, like in the initial five-axis G-code (Code 2).

```
LIN {X 1328, Y 24, Z 968, A 0, B 90, C 0, E1 0, E2 0,
E3 0, E4 0} C_DIS
```

Code 2: Syntax for a linear movement in KRL.

LIN defines the linear movement while the brackets contain an E6POS structure, with the E5, E6, S, and T values omitted. C_DIS declares a blended movement, similar to the G64 command in G-code.

To define a coordinate system that can then be expressed through XYZABC, it is necessary to provide more information than the XYZIJK used to describe a five-axis position, as a coordinate system not only consists of an origin point (XYZ) and a normal vector but can also rotate around that normal vector. In the case of CNC milling, that rotation has no impact on the milling finish – as the tool itself is rotating and rotationally symmetric – but only on the robot's reachability.

In the visual programming environment, the tool center point and the tool axis are again defined by two curves that span a ruling surface, whose rulings then make up the tool axis. The rotation around the tool axis is defined through a Cartesian position that is referred to as the "Orientation Point." A vector is constructed from the coordinate system's origin to the orientation point, reversed and unitized, so the coordinate system's X-axis can be aligned. The process of calculating the XYZABC values is outlined in the appendix.

With the resulting XYZABC values, the linear movement can be assembled. Similar to the G0 (rapid), G1 (linear), and G2/3 (circular) movements of G-code, KUKA robots also allow additional movement types, defining how the robot interpolates between given XYZABC coordinate frames:

- Linear (LIN) movements
- Point-to-Point (PTP) movements
- Circular (CIRC) movements
- Spline (SPL) movements

More recent KUKA controllers also support movements such as SLIN and SPTP, which use a more modern path-planning solution. Choosing an optimum movement is crucial for an efficient toolpath design, as even modern KUKA robots encounter memory limits at around 30MB. Figure 13 shows how a toolpath can be approximated through different movement commands, resulting in vastly different file sizes.

Figure 13: Free-form curve approximated via spline, circular, and linear robot movements.

Circular movements create an arc segment from the robot's current position through an auxiliary point to a given target frame. Unlike G-code, there is no switch between clockwise and counter-clockwise rotation, as the shortest path is chosen through the three points; however, it is possible to define the orientation strategy, whether the tool keeps its orientation from the starting point, blends the orientation along the curve within the plane defined by the start, auxiliary and endpoint, or blends smoothly between the start and end orientation.

Another motion type is Spline movements (Code 3), which can smoothly blend an arbitrary number of XYZABC frames. However, no specific algorithm for the spline type is provided by the KUKA documentation –

the spline is calculated in real-time to provide an optimized toolpath, taking the current motor values, robot position, etc., into account. The splines are, therefore, informed by machinic values rather than relying solely on geometric algorithms. Other controllers like KUKA Sunrise (Chapter 4.7) allow users to select a geometric spline strategy.

```
1 LIN {X -19.812, Y 14.621, Z 0, A 0, B 90, C 0}
C_DIS
2 $ORI_TYPE=#VAR
3 CIRC {X -7.309, Y -10.627, Z 0, A 0, B 90, C 0},{X
15.986, Y -18.459, Z 0, A 0, B 90, C 0} C_DIS
4 LIN {X -19.812, Y 14.621, Z 0, A 0, B 90, C 0}
C_DIS
5 SPLINE
6 SPL {X -7.309, Y -10.627, Z 0, A 0, B 90, C 0} WITH
$ORI_TYPE = #VAR
7 SPL {X 15.986, Y -18.459, Z 0, A 0, B 90, C 0} WITH
$ORI_TYPE = #VAR
8 ENDSPLINE
```

Code 3: KRL code containing linear, circular, and spline movements.

The second group of movements are Point-to-Point (PTP) movements, which provide the most efficient way for the robot to reach a given position. To do so, the robot does not follow a straight path but interpolates between the current axis position and the target axis position, by default, with the slowest axis setting the speed so that all axes arrive at their target position simultaneously. PTP movements can be defined both with E6AXIS values as well as with E6POS (Code 4) values, in which case the robot controller calculates the corresponding E6AXIS value via inverse kinematics.

```
1 PTP {A1 5, A2 -90, A3 100, A4 5, A5 10, A6 -5, E1
0, E2 0, E3 0, E4 0} C_PTP
2 PTP {X 0, Y 0, Z 50, A 0, B 90, C 0, E1 0, E2 0, E3
0, E4 0, S 'B 110'} C_PTP
```

Code 4: Axis-based and Cartesian PTP movements.

This allows the programming of a KUKA robot directly out of a visual programming environment without requiring additional software.

3.3 Kinematics for Robotic Arms

The approaches presented above represent a significant step towards integrating robotic arms into a visual programming environment. However, they might be more accurately described as “scripts” rather than a “program” or a “plugin,” as they automate specific steps but mostly rearrange and transform given data. A central aspect that is missing is a usable simulation, as the approach presented in Chapter 3.2 only allows the user to accurately simulate the movement of the robot’s tool (and the robot’s sixth axis being directly connected with the tool).

As such, one can check for collisions between the tool and the geometry being processed but cannot provide any insight regarding the robot’s position. This makes a reliable simulation of the robotic arm a core requirement for integrating these machines into a visual programming environment so that the user can experiment with a virtual robot before loading a program onto an actual robot.

Doing so requires a kinematic solver for the individual robot. Such a solver needs to be able to perform both forward and inverse kinematics calculations. In the case of forward kinematics, the axis position of the robot is known, and the algorithm needs to calculate the position and orientation of the robot’s tool center point (TCP) based on that information. Inverse kinematics reverts the process in that the position and orientation of the TCP are given, and the axis positions need to be calculated. As such, inverse kinematics is applied to Cartesian movement types like LIN and CIRC, while forward kinematics is applied to axis movements like PTP (see Chapter 3.2.3).

The challenge when working with industrial robots is that their kinematic solver is not documented and available to users, partly because companies consider it a central part of their research and development effort but also because these solvers are generally running in real-time on the robot, reacting to parameters like the current draw of the motors, the encoders of the axes, etc., and are therefore complex to emulate with a non-real-time algorithm. For that reason, the kinematic

simulation within KUKA's own simulation environment KUKA.SimPro is considered only a (close) approximation regarding kinematics and cycle time; for a more accurate simulation, it must be connected with KUKA Office.Lite, which contains an entire robot controller running within a virtual machine. Available generic kinematic solvers like IKFast in OpenRAVE (Diankov 2010) provide capabilities that go far beyond the six to seven degrees of freedom of a robotic arm, adding unnecessary complexity and dependencies within the scope of this research.

3.3.1 Kinematic Robot Definition

The first step towards solving kinematics is the definition of the robot itself. Generally, this is done through Denavit-Hartenberg parameters (Denavit and Hartenberg 1955), a system capable of describing highly complex robotic systems.

As the scope of KUKA|prc is limited to six-axis robotic arms, it only uses an array of six numbers for the geometric robot definition supplemented by the axis velocity and range values provided by KUKA for each robot model. The axes of a KUKA robot are numbered from 1 to 6, with 6 being the robot's flange. As in KRL, they are abbreviated as A1-A6 (Figure 14).

Assuming a robot is standing in its base position with A2 and A3 at a right angle. The base at the origin of a coordinate system, where X is the robot's orientation and Z its vertical height, the first two values describe the offset in X and the offset in Z from the robot's origin to its A2. The third value describes the Z-offset between A2 and A3. For KUKA robots, A4, A5, and A6 are in a straight line in the robot's base position. The fourth value describes the offset in Z between A3 and the extension of the line from A4 to A6. Finally, the fifth and sixth values define the offset in X between A3 and A5, as well as between A5 and A6.

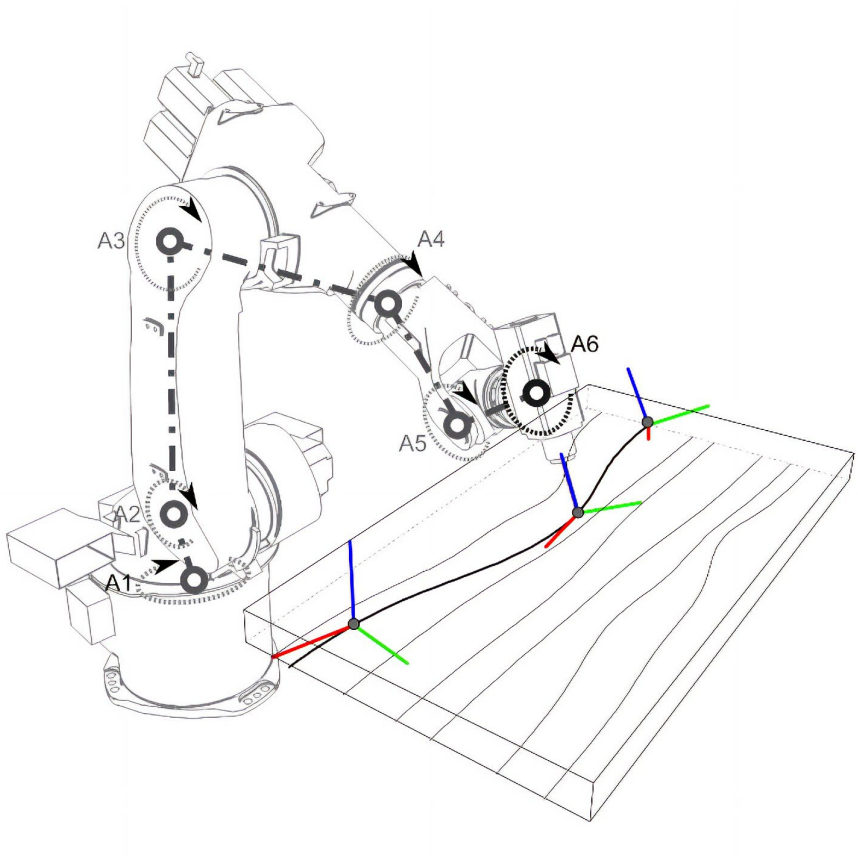

Figure 14: Six axes of a KUKA robot.

In addition, the robot's tool is described as its own coordinate system, defined via XYZABC in relation to the flange of the robot, whose positive Z axis is the normal of the flange and positive X axis is facing "vertically downwards."

By default, a position is set in relation to the robot's global zero point, located centrally in the robot's base. However, it is also possible to use a local coordinate system, a so-called base. This base is again a coordinate system defined via XYZABC in relation to the robot's global zero point, with its positive Z axis facing "upwards" and its positive X axis facing horizontally towards the "front" of the robot.

While forward kinematics can be solved using simple transformations within a geometric library, the mathematics of inverse kinematics are

more complex but well-documented (C. Lee and Ziegler 1984; Mayer, Nagy, and Knoll 2004).

3.3.2 Forward Kinematics

Given the kinematic information of the robot and the offset between the robot's flange and the tooltip, it is possible to calculate the TCP based on the robot's axis values. Each joint is assigned a coordinate system, with the normal parallel to the joint's axis. The tool coordinate system is offset from the flange coordinate system according to the tool's XYZABC values. Then, the rotational transformations, with the coordinate systems' normal as the axes of rotation, are performed in sequence, with each rotation impacting the coordinate systems that follow in the kinematic chain.

Ultimately, the inverse transformation from the robot's base coordinate system is performed – if it is different from the robot's root coordinate system– so that the XYZABC values of the tool frame can be calculated as per Chapter 3.2.

3.3.3 Inverse Kinematics

Inverse kinematics aims to calculate the axis values based on the Cartesian position of the tool. The mathematics for solving a generic six-axis robotic arm are well explained in a paper by Mayer et al. (2004); as such, this section only provides a high-level overview of the approach to solving inverse kinematics.

Commonly, the robot is "split" into two parts, with the robot's wrist (A5) acting as the point of separation. The concept behind that is to solve the values of the initial three axes through basic trigonometry and then deal with A4 to A6 as a single ball joint, as the axes of A4-A6 intersect at a single point (Figure 15).

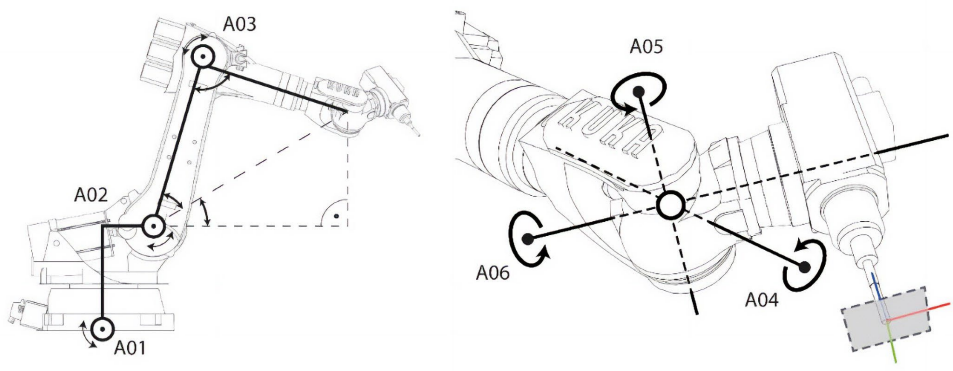

Figure 15: Inverse kinematic system of a six-axis robotic arm.

As the spatial relationship between the wrist point and the TCP is fixed, the robot's measurements are known, and the Cartesian XYZABC position of the TCP is given for inverse kinematics; one can calculate the position and orientation of the wrist point. The value of A1 is simply the angle between the robot's global X-vector and a vector leading from the robot's origin point to the wrist point, projected to an XY plane at the robot's origin point.

For calculating the A2 and A3 angles, a right triangle is constructed so that the three-sided formula for triangles with oblique angles can be used to calculate the lengths of the triangle. By calculating the radius of the inscribed circle and the sum of all lengths, the angles A2 and A3 can be established.

The calculation for A4, A5, and A6 treats the intersection point of those axes as a single ball joint. As such, the current coordinate system of the intersection point after the A1-A3 transformations and the target coordinate system from the given input data are known. Therefore, it is possible to calculate the homogenous transformation matrix between the two coordinate systems and apply the formula for Euler Z-Y-X angles.

3.3.4 Fitting to a KUKA robot

Through the inverse kinematic solver, the values for A1-A6 are now known. However, an important consideration is that the idealized, mathematical model needs to be adapted to the physical robot model's units, motor directions, and other properties.

A critical aspect is the posture of the robot. Unlike a regular five-axis CNC machine that provides only a single way to approach a given position, a six-axis robot has several kinematic configurations where the robot's tool is still at the same position and orientation. In the case of KUKA robots, these postures are expressed through three bits, the so-called Status values (KUKA 2014).

Bit 0 defines the position of the "virtual" ball joint in relation to the rest of the robot. If its position in X is positive, it is considered to be in the "basic" area, and bit 0 is FALSE, while a negative position "behind" the robot is referred to as the "overhead" area. Bit 1 depends on the value of axis 3, with bit 1 being FALSE with A3 < 0. Finally, bit 2 specifies the orientation of A5, setting whether A5 is tilted upwards (TRUE) or is level/tilted downwards (FALSE). This does not refer to the absolute axis value of A5 but depends also on the rotation of A4. In practice, bit 2 flips the direction of A4, often helping with reachability issues that are caused by the limited range of the A4 axis, especially for robots with feed-through IOs like the KUKA KR Agilus series.

Bits 0 and 1 require slight changes to the trigonometric system calculating the initial three axes, changing the signs of several values due to the robot's different orientation compared to the standard model.

Bit 2 only has an impact on the calculation of A4-A6, requiring us to add 180 to the value of A4 and A6 and inverse A5 when the bit is FALSE.

To fit the mathematical system to the actual robot, the values provided by the mathematical library in radians need to be converted to degrees. Offsets are added to fit the rotational direction and zero-position of the mechanical robot.

To provide a non-ambiguous system, KUKA also provides an additional bit for each axis that is TRUE if the axis value is below 0 and FALSE otherwise, referred to as the Turn value. However, as the Turn value does not change the robot's posture, it does not immediately impact the posture itself. It just adds or subtracts 360 degrees and is thus primarily relevant to dealing with axes with a range of over 360 degrees.

3.4 User Interaction Concept

The kinematic algorithms and code generation logic make up the mathematical core of the developed software. However, the software architecture and general user experience are equally important, which was highly influenced by how Grasshopper implemented various visual programming concepts. It is, therefore, essential to provide a more thorough explanation of visual programming in Grasshopper.

3.4.1 Flow-Based Visual Programming

Grasshopper – initially published as Explicit History - has been developed by David Rutten as a visual programming environment for the CAD software Rhinoceros 3D. Rhinoceros 3D results from an interdisciplinary effort between McNeel and Applied Geometry to bring NURBS modeling to AutoCAD and, after several years of beta releases, became a commercial stand-alone software in 1998 (McNeel 2020b).

Due to that history, Rhinoceros 3D has followed the approach of direct modeling, where geometry is drawn and constructed using different geometric tools, ranging from setting control points of the NURBS curve to fitting NURBS surfaces through other arbitrary geometry. After geometry has been constructed, it can still be modified by, e.g., interacting with its control points or – more recently - directly “pushing and pulling” on surfaces and other geometries (Ault and Phillips 2016).

That approach to 3D modeling is a more direct translation of 2D drafting and makes the process comparably easy and intuitive to understand, thus appealing to designers and other creative users. However, it limits the model’s flexibility after it has been created, requiring the user to, e.g., maintain the source geometry out of which the final surfaces were generated (Kuang-Hua 2014).

The second significant approach to modeling is history-based parametric modeling (Thorsteinsson and Page 2006; Shah 2001), initially presented by Pro/ENGINEER in 1987 and since adopted by a wide range of products such as Solidworks, Inventor, CATIA, and NX. Rather than constructing 3D geometry directly, the user defines 2D sketches and certain dimensional, angular, and other constraints, thus building up a

history tree whose parameters can be changed at any time – though it is not guaranteed that all parameters lead to a valid result.

The earliest example is Sketchpad (Sutherland 1964), where a shape is generated through a set of constraints. Compared with direct modeling, history-based parametric modeling needs significantly more advanced planning and, therefore, lends itself less well to an intuitive or playful interaction with geometry. In exchange, it dramatically facilitates changes in later design stages. Simple details, such as fine-tuning the radius of a chamfer operation, can lead to significant work in a direct modeling environment. In contrast, in history-based environments, the user needs to change the radius parameter in the history tree.

While Rhinoceros 3D has introduced a Record History feature that allows some changes after a geometry has been modeled, it does not offer more complex, parametric design capability. However, one of the strategic approaches of Rhinoceros 3D is the support of external developers through a very accessible and powerful set of libraries that facilitate the development of plugins towards adding functionality that goes beyond the scope of the software.

Grasshopper is such a plugin that started as an external, enthusiast-driven project and has since been natively integrated into the software. Grasshopper is a visual programming environment that allows the user to define parametric relationships between objects that go far beyond standard history-based parametric modeling. The concept of flow-based programming (Morrison 2010) is that the programming process does not happen in a textual way but in a graphical way. This often results in a flowchart-like appearance, where modules containing certain functionality or processes are connected via lines or arrows, thus defining their parametric relationship.

Within the area of robotics, a primary visual programming language for PLCs is the Function Block Diagram (FBD), which is standardized as part of IEC 1131-3 (Maslar 1996) and allows the user, e.g., to create individual functions via ST (Structured Text) and then make them more easily usable via the graphical representation as an FBD.

More recently, there have been numerous new visual programming environments presented within the scope of industrial robots, such as

drag&bot (Naumann et al. 2006), Fox|Core by Faude, and the Desk software used to program Franka Emika robots, all of which are running within the browser. Within the greater area of robotic research there are also educational environments like Scratch (Resnick et al. 2009) and Blockly, whose block-based visual programming has been expanded to include industrial robots (Mateo et al. 2014; Trower and Gray 2015; Weintrop et al. 2018).

These environments are highly flexible and can be used to program various tasks, even allowing the easy integration of a wide array of external sensors and data sources. However, their primary use cases can be found in the area of pick-and-place, bin picking, and assembly, as demonstrated by their developers' webpages, as they lack CAD integration and more complex geometric functionality that would be needed to, e.g., derive toolpaths from an imported surface geometry.

Geometry-focused approaches towards visual programming are instead found in the area of real-time visualization and digital art - such as VVVV (2020) and MaxMSP (2020) - and game development – such as Unreal Blueprints (2020) and Unity Bolt (2020). In the area of 3D modeling, visual programming has been pioneered for the definition of photorealistic materials, defining how different textures are blended and linked via a node-based system. Still, it has since been expanded to geometric operations, e.g., via Blender Animation Nodes (Lucke 2020), XPresso (2020b), Houdini (2020), and others.

A fundamental difference in these systems is whether they are real-time-based or not. Programs like VVVV, but also FBDs in PLCs, are generally running in real-time, i.e., the graph is constantly being refreshed at a high rate, ideally at 60Hz for live visuals and potentially much more frequently for PLC programs that might be triggered every millisecond or less, for 1000Hz and more. Grasshopper, on the other hand, is optimized for longer-running, more complex processes and, by default, not constantly updating. Implementing real-time processes into Grasshopper can, therefore, be challenging (see Chapter 4.6)

3.4.2 Grasshopper for Flow-Based Visual Programming

As a plugin, Grasshopper depends on its host software, Rhinoceros 3D, and its underlying geometric library, which has partly been published through the openNURBS initiative (2020). RhinoCommon provides a cross-platform toolkit for plugins to interact with Rhinoceros 3D and its viewport.

To the user, Grasshopper runs in a separate window from Rhinoceros 3D but can access data from the CAD software and draw geometry into its 3D viewport window. The core window consists of a top bar containing the so-called "components" offered by Grasshopper and its various plugins and the "canvas" below, where components can be dragged and dropped to define the data-flow model.

A core aspect of Grasshopper's approach to visual programming is the immediate response. Whenever an upstream component changes, the downstream components that directly depend on it update themselves automatically and display the result without requiring a manual build process as in common. This leads to a very intuitive, experimentation-friendly approach to programming, as changes to the code are immediately reflected by the output(s) of the graph. However, with increasing complexity, the calculation time increases. While the compilation process of text-based programming is generally accepted to take several seconds or even minutes, more complex calculations in Grasshopper can disrupt the user experience. The exact amount of acceptable latency is an area of research within Human-Computer Interaction (Attig et al. 2017). While no general guidelines exist, a latency of around 1 second is generally considered the threshold before an interaction becomes annoying (Tolia, Andersen, and Satyanarayanan 2006). However, for visual effects such as touchscreen interaction, that limit can be as low as 150ms (Kaaresoja, Brewster, and Lantz 2014).

A related criticism of flow-based visual programming in Grasshopper by Davis et al. (2011) is the lack of modularization to facilitate efficient code reuse and the infrequent use of clearly named parameters. While this functionality is included in Grasshopper, Davis et al.'s sample of 2000 scripts showed that only 56% used named parameters and 2.5% contained modules/clusters.

Based on these findings, the following goals were set for KUKA|prc's user experience in Grasshopper:

- Components should be sufficiently responsive to ensure an intuitive user interaction.
- It is essential to perform thorough error checking on the input data and appropriately color-code the component when warnings/errors occur.
- Inputs and outputs should be limited to a reasonable amount not to confuse the user. Stock components rarely contain more than five ports per input/output side.
- Wherever possible, standard Grasshopper data types should be used to ensure an easy data exchange with other plugins within the Grasshopper ecosystem.
- Complex operations should be split into a reasonable number of components so that components are not overly focused but can be reused for many applications.

3.5 User Interaction

The user interacts with KUKA|prc through its over 80 components: 35 components that provide robot-specific functions and a library with 50 more components that contain the kinematic and geometric definitions of various robot models and robotic tools (Figure 16).

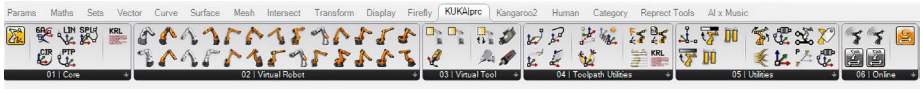

Figure 16: KUKA|prc component categories in Grasshopper.

3.5.1 User Interaction via Grasshopper Components

It has been a primary concern to follow the Grasshopper guidelines identified in Chapter 3.4 and thus only use the minimum number of ports. Everything is arrayed around the prominently placed *Core* component, which acts as the central processing component through the KUKAprcCore.dll library.

It provides five core input ports for the normalized Simulation Slider, the KUKA|prc Commands, the Tool and Robot definition, and Collision Geometry in mesh format. All other values - currently 75 parameters - are contained in a separate graphical user interface. However, it is possible to enable additional ports for externally setting the base coordinates and project name and manually toggle the KRL file generation.

The robot's different interpolation methods are exposed through a series of movement commands for Axis, LIN, PTP, CIR, and SPL movements, which generally consist of one set of inputs for defining the target position, either via six axis values or through one (LIN, PTP) or more (CIR, SPL) robot frames, which in Grasshopper are represented by the *Plane* geometry type. Each movement command also contains an input port for setting the speed, either in percent for PTP/Axis movements or in m/sec for LIN, CIR, and SPL movements.

Additional functionality, such as setting the type of motion blending or activating the input ports for setting the values for up to four external axes (E1-E4), are placed in a context menu to reduce the number of input ports. The *Custom KRL* component can insert arbitrary KRL code into the KRL file generated by KUKA|prc and has a single input port for `strings`.

The robot's tool can be defined via a numeric input of the previously calibrated XYZABC values or a referenced plane geometry. In both cases, tool geometries placed at the global origin of the CAD system are transformed to the robot's flange.

Robot models are grouped into families of geometrically identical robots; e.g., a KR6R1100 robot shares its model with a KR10R1100, as their only difference is the motor parameterization. The robot may be linked with additional axes by plugging the output of a robot model into the input of an external axis. A *Custom Robot* component allows users to modify existing robots' parameters or define completely new machines by setting their names, geometry as meshes, kinematic layout, lower and upper axis limits, and axis speed. The *Robotic Cell* component enables users to implement peripheral geometry without changing the robot model itself.

While these components cover all essential functionalities, additional tools allow the user to rapidly add safety movements to a robotic toolpath through Cartesian values or offsets along the tangent or normal of its path. Another function serves to weave sets of robot commands and group or ungroup them.

Instrumental is the group of import tools that provide the option to import toolpath data from other specialized software. At the moment, KUKA|prc contains import functionality for the four formats: five-axis G-code generated in Fusion 360 and exported via a custom-developed postprocessor in Extensible Markup Language (XML), standard DIN 66025 5-axis G-code (2020), three-axis G-code for 3D printing, and KRL import, which is optimized for re-importing KRL code that was previously generated through KUKA|prc.

The requirement for three separate G-code import components already demonstrates the challenges of working within a system that is - to the outside - following a given standard like DIN 66025. Still, in practice, every vendor implements custom functionality and systems.

Each component also requires additional input data beyond what is provided by the G-code to generate a valid, plausible robot program. As such, the user needs to define a rapid speed that gets assigned to G0 commands and set an orientation point, which defines the rotation around the tool axis towards a given point in space (see Chapter 3.2). Furthermore, G-code can be very dense and complex, causing robots to quickly run into memory limits, unlike dedicated CNC machines.

Therefore, the user can define a Cartesian tolerance in millimeters and an angle tolerance in degrees so that an algorithm can discard the least relevant positions accordingly. The *Analysis* component is essential for the (automated) optimization of toolpaths as it provides numeric outputs for the calculated process times, axes values, etc., coming from the *Core* component.

KUKA|prc also provides two interfaces for remote controlling KUKA robots in real-time:

- KUKA.mxAutomation for KRC4 robots
- KUKA.smartServo for Sunrise-based collaborative robotics

The *Sunrise Communicator* uses the KUKA.smartServo functionality on the iiwa for moving the robot, communicating via UDP datagrams (see also Chapter 4.7). As an output, the component provides the current axis and Cartesian position, the force at the tool's tip as a vector, and the number of buffered commands.

mxAutomation is used to achieve a similar level of control on KRC4 robots (see also Chapter 4.6) but supports significantly more features, which are therefore exposed through a graphical user interface. The component provides a single output that can be connected to the *mxA Display* component, which visualizes the robot and outputs the robot's position, IO-state, and various debugging information in real time.

3.5.2 Graphical User Interface

The user can access the KUKA|prc graphical user interface from within the chosen host software, i.e., unlike programs like RoboDK that transfer toolpath data into their simulation environment. It has evolved from a task-specific platform for flank milling (Brell-Cokcan et al. 2009) into a more universal, streamlined user interface that supports various applications.

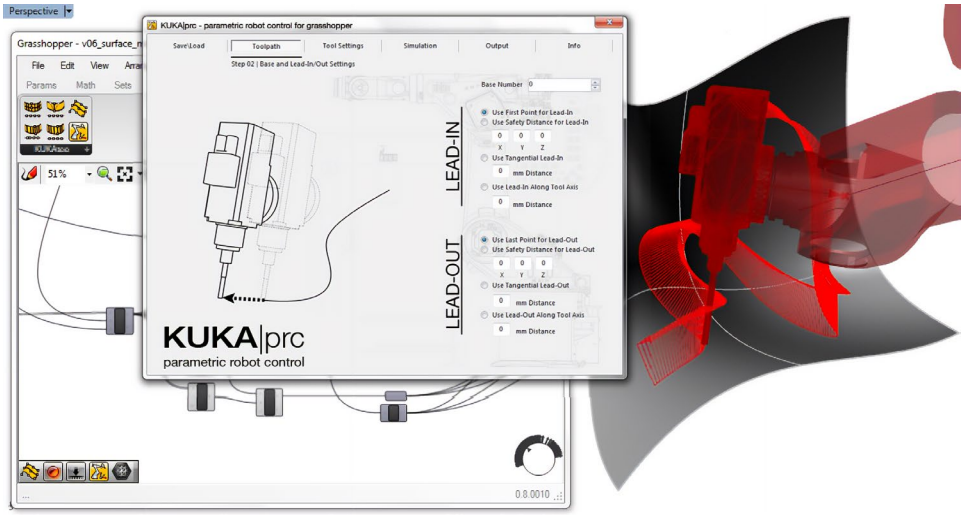

Figure 17: Early version of KUKA|prc, 2011.

It is structured into three tabs, Settings, Advanced, and Analysis, with the settings laid out in a way that is structured by how frequently the average user needs to change a parameter (Figure 18). As such, the most critical parameters are immediately available on the initial "Settings" tab: Project name and output folder, default PTP and LIN speed, Base number and XYZABC values, and a slider to adjust simulation fidelity. For more complex projects, the user can change to the "Advanced" tab, where the most relevant options are on the first tab, with three more sub-tabs providing more seldomly used options.

The initial "Advanced" tab provides settings regarding the start and end position of the robot, defined as axis values, the toolpath-interpolation parameters, and in-depth simulation options. Further tabs allow the user to set up an automatic solver for the external axes and change the code generation from standard KRL to CAMRob, KUKA.CNC G-code or Sunrise Java code, as well as modify the code generation itself.

Finally, the "License" tab installs and verifies license files. Every section of the GUI comes with a help button in the upper-right corner that explains the effect of the individual settings.

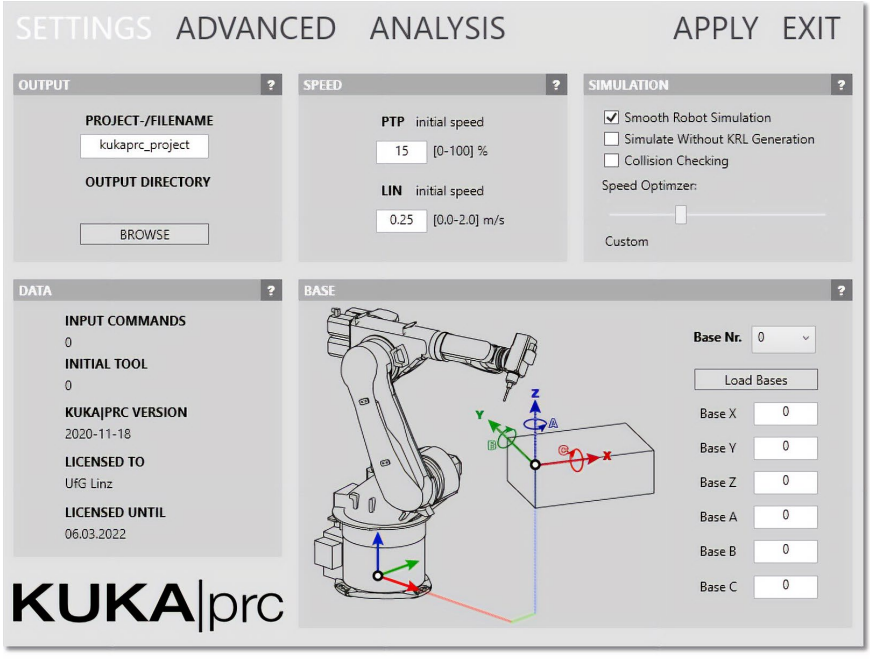

Figure 18: KUKA|prc graphical user interface.

The “Analysis” type is of specific importance to the user experience, as it plots the robot’s axis movement over time, with time on the horizontal X-scale and the axis value on the vertical Y-scale. By default, the axis values are absolute. However, they can also be normalized to range from -1.0 to 1.0 to see more easily when approaching an axis limit.

Each axis is given its own line plot. The default background is grey but changes to red for collisions and unreachable positions and to yellow for positions that may be singularities. A vertical white bar signifies the current position, and the axis values at that position are shown numerically on the right. In the case of an unreachable position or a collision at that axis, the number is colored red.

Clicking any position on the graphs updates the currently simulated position, showing the robot at the chosen position. A new input from the simulation slider overrides that simulation position.

The graphical user interface covers more than 75 parameters, grouped intelligently to minimize the user’s time to access the desired parameter.

The Appendix section presents two exemplary workflows demonstrating the process of programming a robot within Grasshopper.

3.5.3 Interaction Analysis

KUKA|prc’s 85 components provide many options; however, in practice, it can be observed that just a few essential components are frequently used, as most algorithms rely primarily on “generic” Grasshopper programming using stock components and then using the KUKA|prc components primarily for turning plane-geometry into movement commands and simulating/optimizing the resulting toolpaths.

Beyond anecdotal experience, this can be observed in the workflow examples above and is also supported by data analysis. A program was developed that first extracts all GUIDs – unique identifiers like {11a46c3d-20c1-4a57-b704-bce0ea919cef} – and names of the KUKA|prc components from the software’s source code. It then counts the matches in Grasshopper GHX files, which use an XML structure internally. As Grasshopper, by default, saves binary GH files, a separate conversion

step was integrated through Grasshopper's GH_IO.dll library, converting binary files to the easy-to-read XML format.

There is no official documentation of Grasshopper's XML schema; therefore, it was necessary to analyze its data structure. Each GHX file contains a series of so-called chunks containing so-called items. For example, a robot component is represented as a chunk named "Object," containing three items and another chunk. The items define the GUID of the component, the GUID of the library, and the component's name. The chunk contains another chunk referred to as a "Container," again with chunks and items.

As such, <chunks> serve to structure the data and provide only small amounts of metadata, while each <items> node contains several <item> nodes, which in turn contain various data types and metadata (Code 5).

```
<chunks count="1">
    <chunk name="Attributes">
        <items count="2">
            <item name="Bounds"
type_name="gh_drawing_rectanglef" type_code="35">
                <X>257</X>
                <Y>308</Y>
                <W>189</W>
                <H>24</H>
            </item>
            <item name="Pivot"
type_name="gh_drawing_pointf" type_code="31">
                <X>351.5</X>
                <Y>320</Y>
            </item>
        </items>
    </chunk>
</chunks>
```

Code 5: Part of the XML data contained within a Grasshopper GHX file.

A custom tool then deserialized the XML files and extracted the information referring to the components used in each Grasshopper file.

To filter out Grasshopper files that may not be related to KUKA|prc, only files containing at least one KUKA|prc *Core* component were considered. A total of 720 Grasshopper files were taken from user requests, the Robots in Architecture forum, and the author's research and teaching materials.

The counting process was done twice, once for the total number of occurrences and once for counting just the first occurrence per file (Figure 19). As the total count can be easily skewed by files that contain an unusually large number of component instances, the single occurrence count produces a more representative number. This can be seen, e.g., in the *Synchronize Robot* component, which is used 827 times in just 21 files, making it the fourth-most used component for total usage but just the 32nd-most used component for individual files.

As the filter removed all files without a KUKA|prc *Core* component, the *Core* component is present in all 720 robot projects (100%), followed by the *LINear Movement* component with 547 projects (76%) and *Custom Tool* with 437 (61%) projects. After the top three, there is a steep drop in frequency so that only the following eight components reach more than 10%: *PTP Movement* (30%), *Orient Plane* (29%), *AXIS Movement* (27%), *Tool-Axis Offset* (26%), *Analysis* (19%), *KR6-16-20 Robot* (15%), *Agilus KR6-10 R1100 Robot* (13%) and the *Play* Component (11%).

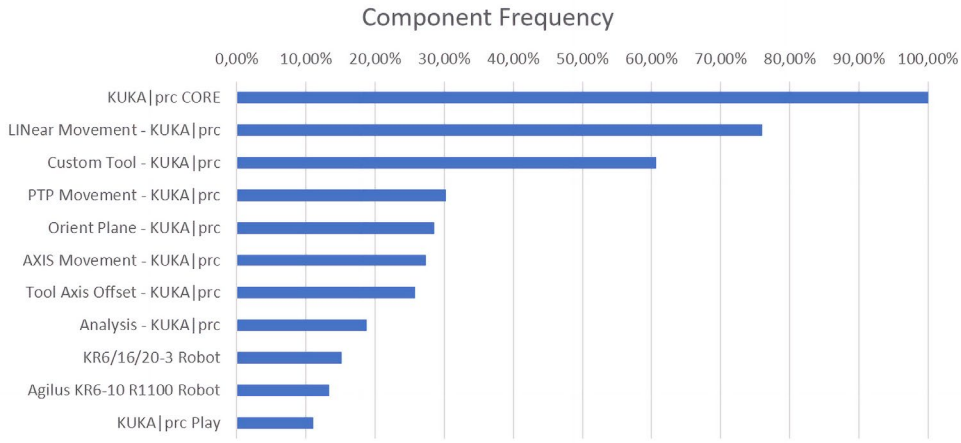

Figure 19: Component frequency in 720 analyzed Grasshopper definitions.

The analysis of the Grasshopper components clearly shows that users depend on a few popular components, which are sufficient to realize even complex projects. In contrast, most components are rarely used for specialized purposes.

As the files contain many of the author's work, the data is possibly biased. Analyzing just the 185 Grasshopper files originating from the Robots in Architecture forum, the top components were identical with *Core* (100%), *LINear Movement* (81%), and *Custom Tool* (53%). However, *PTP* (13%) and *AXIS* (9%) Movements are significantly less popular with the broader range of forum users.

The data collected should still represent an accurate overview of KUKA|prc usage and the most commonly used components. This means that users do not need to understand the fine details of dozens of components but only internalize a select few workflows and processes to realize robotic processes, thus significantly opening up robotics for new users, ideally building upon an existing understanding of geometry.

4 Software Architecture for Parametric Robot Control

The unique selling proposition of KUKA|prc and its underlying flow-based visual programming environments is that it constrains robotic movement to geometry, allowing new applications that would otherwise not be feasible with other approaches (Figure 20).

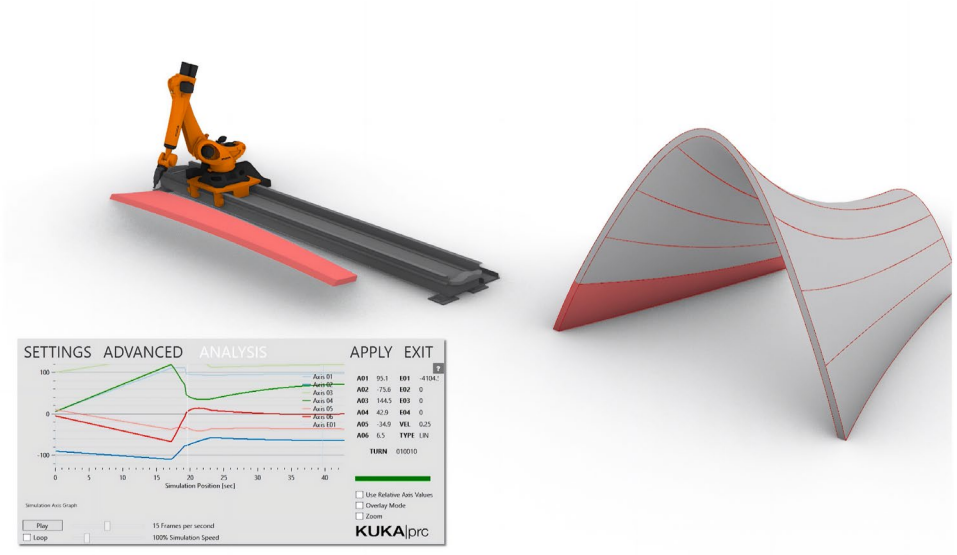

Figure 20: Robotic toolpath constrained to a parametric geometry for enabling mass customization.

This approach of embedding robot control software in several flow-based visual programming environments leads to several challenges that need to be approached through a highly customized software architecture.

While software development generally addresses the underlying software architecture before the user interface, KUKA|prc's approach is strongly guided by the potential and constraints of the visual programming environments. This is, therefore, reflected in the structure of this thesis, with the software architecture following up on the user experience.

The following chapters present the structure of the software and its interaction with different programming environments as well as robot controller architectures.

- The general structure of the project and its separate libraries is presented in Chapter 4.1.
- Chapter 4.2 deals with the challenge of processing geometry in different software environments, namely customizing one's geometry library rather than relying on native implementations.
- The data flows for robot simulation are presented in Chapter 4.3.
- In Chapter 4.4, the sequence of operations from input data to reliable simulation is discussed, with a special consideration of the constraints of flow-based programming and the sequential nature of robot programs.
- Details on the individual integrations of KUKA|prc into flow-based programming environments, as well as exemplary console and game engine integrations, are provided in Chapter 4.5.
- Chapter 4.6 goes beyond offline programming and presents mxAutomation as an interface to control robots in real time from a flow-based programming environment.
- As an outlook to future robot control architectures, Chapter 4.7 demonstrates how to control new collaborative robots using the Java-based Sunrise architecture.

4.1 Software Layers and Dependencies

As KUKA|prc supports several platforms, programming environments, and operating systems, it is structured at three layers: The universal "Core" layer, the operating-system dependent "OS" layer, and the application-specific "Application" layer. This ensures that a maximum amount of code can be reused, thus making it easier to push updates to all platforms and maintain code.

An initial choice with a significant impact was the programming language itself. The primary constraint in that area was the underlying CAD

software Rhinoceros 3D, which at that time only supported C++ and .NET plugins. Support for Python was only added in 2012 through GhPython, and a JavaScript version of Rhino3dmIO was added in 2019. Grasshopper itself is programmed in .NET and requires the components to be defined as .NET objects extending its GH_Component class, .NET was the best choice for developing a new plugin. Of the four languages that share. NET's Common Language Runtime (CLR), C#, Visual Basic .NET, Managed C++, and JScript.NET, C# was selected as its syntax felt the most natural to the author, being very similar to Java, likely due to their shared heritage (Nagel 2018).

Today, through the open .NET Foundation, the development of .NET (Core) has become an open-source effort focusing on cross-platform applications. Thus, it allows KUKA|prc to cover the Core, OS, and Application layers with a single programming language (Figure 21).

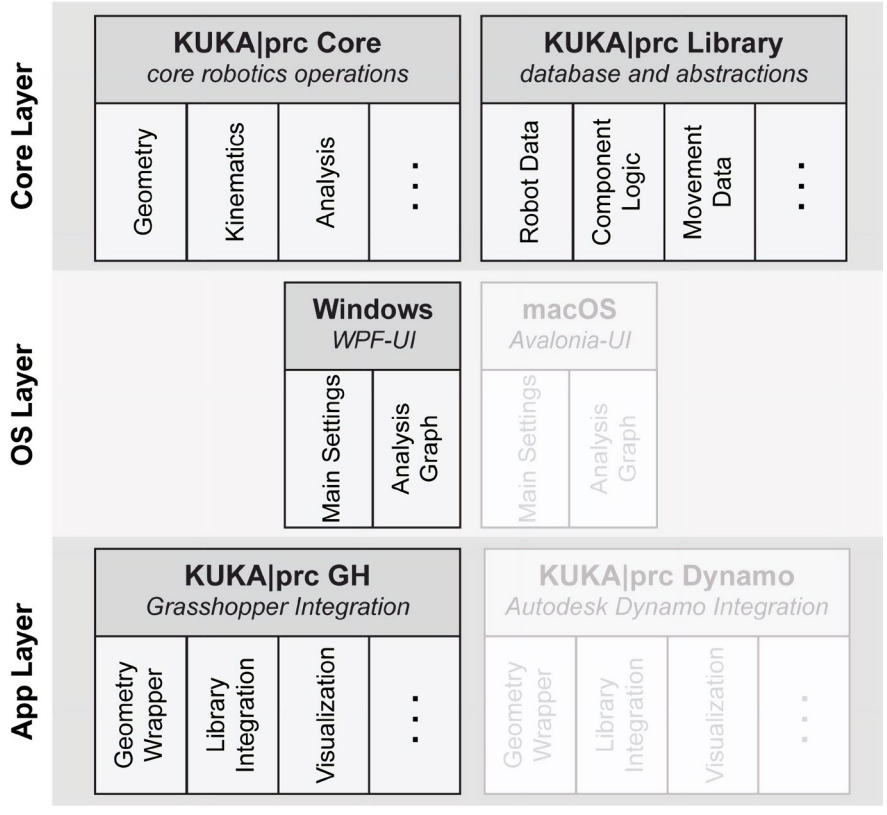

Figure 21: KUKA|prc software layers. Components used for KUKA|prc for Grasshopper are emphasized.

4.1.1 Core Layer

KUKA|prc consists of two main libraries, the KUKAprcCore.dll and the KUKAprcLibrary.dll. KUKAprcCore.dll contains all core logic and algorithms, while the KUKAprcLibrary.dll builds upon that logic to provide the internal functionality of all visual programming components as well as the geometric and kinematic data of the included robots, tools, etc., in a universal format. As such, the KUKAprcLibrary.dll depends on the KUKAprcCore.dll, while it would be possible to work with the KUKAprcCore.dll without the added convenience of the KUKAprcLibrary.dll.

Both libraries are written in .NET and conform to .NET Standard 2.0. The advantage of .NET Standard 2.0 is that the generated libraries are compatible with several platforms within the same file, such as .NET Framework 4.6.1, .NET Core 2.0, and Mono 5.4 (Microsoft 2020). Therefore, KUKA|prc can run on a wide range of operating systems. Nearly all end-users use KUKA|prc through either McNeel Rhinoceros 3D or Autodesk Dynamo; the .NET Framework is the most common platform.

The minimum required version 4.6.1 was released at the end of 2015 and is therefore widely installed among Windows 7 users – whose number is decreasing with the end of Microsoft's support of the platform – while Windows 10 keeps the .NET Framework automatically updated, currently at version 4.8, which is by design fully downwards compatible. In previous research, the Mono platform has allowed prototypical implementations of KUKA|prc to run on Linux-based operating systems like Raspbian. However, future cross-platform and cloud applications are expected to be built upon .NET Core due to its easy portability and high performance.

4.1.2 OS Layer

While KUKA|prc's core logic is already cross-platform capable and running on Windows, Linux, as well as MacOS, the graphical user interface is so far provided on a per-OS basis, as it has not been possible to develop an appealing cross-platform solution.

The author decided to use Microsoft's Windows Presentation Foundation (WPF) as the UI platform. WPF's user interface design tool Blend provided a very designer-centric approach to UI design, similar to vector drawing in Adobe Illustrator. As an added advantage, the OxyPlot library, used for the plotting of the robot movement over time, offered an easy-to-use interface to implement plots into the WPF environment, building upon the Model–view–viewmodel (MVVM) pattern (Garofalo 2011) to easily update and interact with data from the *Core* component.

As such, KUKA|prc's Windows-based UI is contained within the KUKAprcLibraryWIN.dll and is built upon the .NET Framework 4.6.1; therefore, it is not cross-platform capable. In the future, more recent UI options like the .NET Core-based Avalonia UI library – building upon Google's cross-platform Skia platform for drawing – may help bring a shared UI for multiple operating systems, thus making the OS layer irrelevant.

4.1.3 Application Layer

The application layer's task is integrating the application agnostic KUKA|prc logic into proprietary programming environments such as McNeel Grasshopper and Autodesk Dynamo. Therefore, the application layer depends on libraries provided by the developers of the host software, such as Grasshopper.dll and RhinoCommon.dll for the Grasshopper implementation and the various `DynamoVisualProgramming` libraries distributed through the package manager NuGet by Autodesk. Currently, those libraries are distributed as .NET Framework 4.5 libraries on the Windows platform, though at least McNeel also provides versions compatible with the MacOS version of Grasshopper.

The application layer has to interact with the native software naturally and efficiently, often requiring different approaches to result in a comparable user experience between platforms. A primary task is, therefore, to wrap the functionality of the KUKAprcLibrary.dll in native components, considering the UI specifications and dataflow limitations of the host environment.

To do so, the proprietary geometric types of each platform's geometry kernel (`Rhino.Geometry` for Grasshopper and `Autodesk.DesignScript.Geometry` for Dynamo) need to be converted into KUKA|prc's generic, internal geometric definition for computation and then back again into native geometry to be displayed in the software's viewport – a process that needs to be streamlined and optimized as to reduce latency.

Another task is implementing the user interface and 3D drawing process, with the appropriate, thread-safe communication between them. Finally, the application layer needs to implement additional functionality to compensate for the limitations of platforms, such as the minimal support for meshes in Autodesk Dynamo. The Grasshopper application layer is KUKAprc.gha – a renamed DLL library – while in Dynamo, the Application Layer is split into three libraries, the KUKAprcDynamo.dll containing the definition of each node's input and outputs, the KUKAprcDynamo.customization.dll, containing the icons of the nodes, and the KUKAprcDynamoLogic.dll containing the logic that wraps the KUKAprcLibrary.dll. This separation is due to Dynamo's MVVM pattern, where one library implements the `NodeModel` interface -binding data with the Dynamo UI - and the other contains the logic.

4.1.4 External Libraries

The three layers are supported by a small number of additional libraries: OxyPlot, a plotting library used for the Analysis view, with the main functionality within the OxyPlot.dll and the WPF implementation in the OxyPlot.Wpf.dll, and Newtonsoft's Json.NET framework providing serialization and deserialization for settings, robot models, etc. via the Newtonsoft.Json.dll. SharpDX's geometric library (see Chapter 4.2) is embedded in the KUKAprcCore.dll, not a separate DLL.

Microsoft provides the netstandard.dll and remaining libraries, such as System.Collections.dll, primarily for compatibility with the older .NET 4.6.1 framework by forwarding calls to the system assemblies. Targeting a higher version would reduce the number of libraries but possibly limit some users from working with the software.

4.2 Mathematics and Geometry Implementation

4.2.1 openNURBS Geometry

A powerful mathematics and geometric library is a core software component for robotics applications, especially complex kinematics. Initially, KUKA|prc was built entirely on the RhinoCommon framework, using the Rhino-native `Rhino.Geometry` namespace. As RhinoCommon is limited to running within Rhinoceros, more options were needed.

McNeel provides Rhino3dmIO, a feature-reduced version of Rhinoceros 3D's geometric kernel, built upon the openNURBS Toolkit. The primary purpose of Rhino3dmIO is to enable developers to read and write 3DM files, e.g., for importing Rhino geometry into other software packages. As such, many operations for geometry definition, transformation, and analysis exist; however, advanced functionality such as mesh-mesh intersections is omitted. There is currently no feature-to-feature comparison between Rhinoceros 3D and the feature set of Rhino3dmIO.

While the limited feature set is not an immediate problem, the more significant issue is the dependence of Rhino3dmIO on its openNURBS library, which is written in C++, making debugging more complex and requiring new builds for each platform. License-wise, Rhino3dmIO uses the MIT license and, therefore, allows both commercial and non-commercial use and modifications.

4.2.2 Overview of Geometric Libraries

Due to the experience with Rhinoceros' native geometry library, which closely follows the design-centric approach of CAD, the goal was to find a C#-native, accessible geometry library with a permissible license. An evaluation of well-known commercial geometry kernels like C3D (C3D Labs 2020), Parasolid (Siemens 2020), and CGM (Dassault 2020) shows that while they each provide C# bindings, they rely on native, non-managed proprietary libraries and offer a range of functionality that goes far beyond the requirements of KUKA|prc, at an extremely high cost.

Within the scope of open-source software, geometric libraries can be divided into two areas: scientific computing and game engines. A .NET

library within the scientific field is, e.g., Math.NET (2020), which provides functionality for numerical computing, where linear algebra and, by extension, the creation and manipulation of matrices, as well as Math.NET Spatial, which covers the geometric aspects. While it contains core functionality like definitions of coordinate systems, points, vectors, circles, etc., it lacks support for geometries like arcs, curves, and, most importantly, meshes and an easy way to transform and scale geometry.

Therefore, the geometric libraries of game engines may be closer to the requirements of CAD-adjacent software than scientific libraries. Game-focused geometry libraries generally follow the standards established by either OpenGL or Direct3D. A core difference is the handedness of their world-coordinate systems, with OpenGL using a right-handed coordinate system while Direct3D, by default, uses a left-handed coordinate system. Rhinoceros 3D and most CAD software follow a right-handed, Z+ up coordinate system.

In .NET, several wrappers connect .NET languages with graphics libraries such as OpenGL (OpenTK 2020) and Direct3D (SharpDX 2020), a subset of DirectX. The advantage of such wrappers is that they cover a large part of the generic functionality of their graphics libraries. At the same time, actual game engines focus more closely primarily on mesh geometries and their transformations and intersections.

While a large part of either wrapper relies on the native functionality of Direct3D and OpenGL, the basic geometric functionality, such as matrix transformations, is implemented directly in C#. This is probably because calling a native library is inefficient for relatively small operations. As a result, the geometric functionality is independent of the rest of the library, making it possible to implement it into custom software with a reasonable effort and without having native libraries as dependencies.

Evaluating both OpenTK and SharpDX showed SharpDX's architecture closer to RhinoCommon, providing more geometric types and offering more utility functions to transform, intersect, and generally interact with geometry. Therefore, the `SharpDX.Mathematics` namespace was chosen to form the basis of the KUKA|prc geometry system.

4.2.3 Customizing a Geometry Library for Robot Control

`SharpDX.Mathematics` provides highly valuable base functionality concerning matrix operations and specific base geometries. However, as a generic interface, it needed to be customized for KUKA|prc's requirements. The approach was to create a set of classes that wrap SharpDX's geometry classes so that Rhino3dmIO and the SharpDX-based geometric solution could be easily exchanged.

To do so, a `Point3d` class – similar to the `Point3d` class in Rhinoceros 3D – is created, which internally uses the SharpDX `Vector3` class to store the data as three single-precision floating-point numbers. While the SharpDX `Point` class sounds more suitable, it only contains two numbers for XY screen coordinates. Then, the different methods need to be modified. In SharpDX, the `Vector3` class includes a static `Distance` method, which takes two `Vector3s` as arguments (`public static float Distance(Vector3 value1, Vector3 value2)`), Rhinoceros 3D implements the distance as a property (`public double DistanceTo(Point3d pt)`). As such, the KUKA|prc geometry wrapper formats the data in a way that conforms with Rhinoceros 3D on the outside but uses SharpDX internally.

Implementing support for arcs and splines was more challenging than the primary point and vector classes, as SharpDX did not support those geometry types. However, they are crucial for supporting the accordingly named robot motion types. Integrating the mathematical system of NURBS would have required significant work without providing any innovation over, e.g., the existing Rhino geometry kernels. A middle ground was found in integrating algorithms that can represent a freeform curve like a spline and an arc as a list of points, and from then on, only operate on the list of points. A disadvantage of that process is that the curve resolution is fixed as curves become polylines.

Splines on KUKA robots present a unique challenge as they are generated in real-time, based on current motor data and other parameters, to achieve a toolpath that interpolates smoothly through a given set of points. Because of that, no specific algorithm (like NURBS, Akima, Bezier, etc.) can reproduce the robot's spline exactly. For representation purposes, KUKA|prc uses a very straightforward implementation of Catmull-Rom splines (Catmull and Rom 1974) that

splits a list of given points into sets of four points, applies the Catmull-Rom cubic polynomial and then repeats the process until a given point resolution is achieved. The output, also displayed in the toolpath visualization, is a list of points represented as a `Polyline` object. A similar approach is used for arc segments.

The `Plane` class had to be programmed entirely from scratch, as SharpDX's plane class is an infinite plane without orientation, i.e., defined by a point and a normal vector, while planes in Rhinoceros 3D are coordinate systems. Therefore, the KUKA|prc `Plane` class builds upon the SharpDX `Matrix` class, which is a 4x4 matrix by default. A standard constructor in Rhinoceros 3D that defines a plane based on an origin point, an X-vector, and a Y-vector, therefore, needs to be broken down into a series of operations: First, the input vectors are unitized and, through their cross-product, the Z vector calculated. With that input data, 12 of the 16 fields of the matrix are now known, with the last column becoming {0,0,0,1}.

Other constructors needed more elaborate solutions, as their scope is outside that of a regular geometric library. An example is the definition of a plane through a point and a normal vector (see Chapter 3.5), which is geometrically not fully defined. In KUKA|prc, an initial X-vector is determined based on several criteria, such as avoiding the X-vector being parallel to the given Z-vector. One can now calculate the Y-vector through the cross product of the Z and X-vector. To ensure that all vectors are oriented at 90 degrees to each other, the Z now replaces the initially arbitrary X-vector- and Y-vector cross-product.

In addition to the new constructors, an extensive range of properties was defined, e.g., to extract a `Vector3d` of the X-axis from the `Plane` object's underlying `Matrix` class. Some properties also build upon existing SharpDX functionality, such as the `ClosestPoint` method with a `Point3d` argument, searching for the closest position on a plane to a given point. For that, the SharpDX `Plane` class can be used, as the orientation is of no importance, casting a ray from the point along the normal of the SharpDX `Plane` and searching for the intersection – a very typical operation in 3D graphics that is therefore natively supported by SharpDX. Transformations of the KUKA|prc `Plane` class are simple to

implement, as they are just a matrix multiplication between the `Plane`'s underlying matrix and the transformation matrix.

Transformations in SharpDX are performed by the native Direct3D functions; as such, they had to be integrated from the ground up. However, the algorithms are easily found in literature (Glaeser and Stachel 1999) and need to be adjusted to C# and the specific functionality of the SharpDX geometric library. A valuable reference in C# is OpenTK's mathematics namespace, though modifying the code to compensate for the different handedness of the coordinate systems in OpenGL and Direct3D is necessary.

In addition to fundamental transformations such as translation and rotation, transformations remapping coordinate systems are particularly important within the scope of robot kinematics, as the robot moves its tool coordinate system in a local base coordinate system, whose position is set in relation to the robot's world coordinate system. In Rhinoceros 3D, this is done by the `PlaneToPlane` and `ChangeBasis` transformations. Where `PlaneToPlane` moves a geometry from one coordinate system into another – inverting the origin-matrix and then multiplying it with the destination matrix – `ChangeBasis` expresses a position in one coordinate system through another coordinate system, for which it is necessary first to get the transformation from the world coordinate system to the first coordinate system and the transformation from the second coordinate system to the world coordinate system and then to multiply both resulting matrices.

Another missing core feature that had to be implemented was the support of mesh geometries. Following the template of the `Rhino.Geometry.Mesh` class, the KUKA|prc `Mesh` consists of three lists: The vertices of the mesh are expressed as points (`Point3d`), the normals of the mesh as vectors (`Vector3d`), and the faces as sets of four integers (`Int4`) that refer to the indices of the points, making up either a triangular or a quadrilateral face.

While `Rhino.Geometry.Mesh` also contains properties relating to texture mapping and vertex colors; KUKA |prc only uses a single color per mesh. Finally, to optimize the performance of the code, the KUKA|prc `Mesh` also contains a single plane, by default, located at the global origin. When a mesh is transformed, that transformation is only applied to the

single plane object rather than hundreds or thousands of vertices. Transforming the vertices is the last step before a mesh is displayed through the `Bake()` method.

In a robot model, a single joint may often contain multiple meshes with multiple colors, such as the orange, embossed KUKA logo on a white KUKA Agilus' axis 4. Therefore, various meshes can be contained within a single `MeshCollection`, with one `MeshCollection` providing all geometric data for a single robot joint. A `MeshCollection` may also contain two sets of meshes, one with a high-resolution mesh for high-quality visualization and rendering and a second, lower-resolution set of meshes.

4.2.4 Implementing Collision Checking

That low-resolution mesh is essential for collision checking. As the avoidance of collisions is one of the core reasons for the simulation of the toolpath of a robotic arm, any performance improvement translates into an immediate gain in usability. As the `Mesh` class was built from scratch, no functionality for collision checking was implemented by default. The topic of collision detection is highly complex and multi-layered (Ericson 2004) and especially computationally intensive for mesh-mesh collisions.

A brute-force approach would check each triangle against all triangles of the collision geometry. A simple collision shape of around 1000 polygons and an average KUKA low-resolution robot model in KUKA|prc with around 8000 polygons would result in 8 million collision checks for a given position. A 50m long toolpath with an interpolation resolution of 10mm results in 40 billion collision checks.

Therefore, a more efficient approach than brute force calculations is needed. Many geometric libraries actively discourage mesh-mesh intersection and instead support primitives like spheres or capsules. The advantage of such a geometry is that the collision checking can be reduced to a simple proximity calculation. If the closest point on a mesh is closer to the sphere's center than its radius, there is a collision. On the other hand, capsules may be represented by lines, calculating the closest point between a line and a mesh.

For mesh-mesh collisions - as are required for KUKA|prc – a well-published approach is to use AABB (axis-aligned bounding boxes) trees (Wang and Liu 2014) to roughly check whether there is the potential for collisions between geometries. This algorithm constructs bounding boxes and then checks whether they intersect. If there is no intersection, no collision is possible. The algorithm's tree' aspect increases the process's efficiency by hierarchically grouping geometry in trees, also called a bounding volume hierarchy (Gu et al. 2013).

As a result, nodes in that tree that are close to each other are also geometrically close and should be checked for AABB collision. This phase is called the broad phase, followed by the narrow phase, which performs accurate, triangle-based collision checking.

These algorithms do not have to be implemented from scratch; instead, existing physics libraries, which also implement collision checking, may be used. For KUKA|prc, Jitter Physics (Leibowitz 2020) was chosen due to its easy integration into a non-realtime environment. To check for collisions, mesh geometries must be converted into a Jitter Physics mesh representation called `RigidBody`. This takes up considerable computational power as it needs to be performed for every robot position.

In comparison, the `Rhino.Geometry` namespace of Rhinoceros 3D (Version 6) does not offer any dedicated mesh-mesh collisions functionality; however, it can calculate the polyline resulting from the intersection between two meshes. A polyline length higher than zero signifies a collision between two geometries. Benchmarking both Jitter Physics' and the Rhinoceros 3D approach showed Rhinoceros 3D to be considerably faster than the C# implementation.

An exemplary project with a collision geometry of 500 faces and a robot model with 3000 faces checked against 49200 positions takes approximately 925ms (10 measurements, minimum 818ms, maximum 1100ms) to compute using Rhinoceros 3D's native methods. On the other hand, the JitterPhysics implementation takes 17.5 seconds (10 measurements, minimum 17.1 seconds, maximum 17.9 seconds).

Further optimization may potentially improve the JitterPhysics implementation; however, considering the significant performance

delta, the `PRC_CheckCollision` method has been implemented as a virtual method so that it can be overridden by the host software in case the host software offers superior collision-checking algorithms. The custom Jitter Physics implementation is used as a fallback for standalone applications and provides reliable, if slow, results.

4.3 Data Flow for Robot Simulation

4.3.1 Challenges

A significant challenge of flow-based visual programming and robot simulation in general is the data flow. As per Chapter 3.4, Grasshopper is implemented as an acyclic, directed graph so that data flows exclusively from "left" to "right." In KUKA|prc, the main settings are defined in the *Core* component, which also performs all core calculations and the robot simulation. Ideally, the settings would be described as the programming's first ("left") stage and then accessible by every component.

In practice, such a layout is not feasible, as it would require every KUKA|prc component to provide an input port for the KUKA|prc settings component, which would lead to many unnecessary links and not fit in with Grasshopper's concept of a smooth left-to-right dataflow. In a programming context, the settings might be defined as a static object accessible by all components. However, this would limit the environment to a single KUKA|prc *Core* component across all open Grasshopper definitions, as the individual upstream components would not be aware of which – possibly also multiple! – downstream *Core* components they might be connected to.

The placement of the settings/simulation component at the end of a data flow is, therefore, a compromise with several drawbacks; most of all, it becomes hard to allocate computational power intelligently. A robotic fabrication process might contain multiple branches of processes, e.g., a pick-and-place process followed by a subtractive process. Ideally, the simulation, from code generation to inverse/forward kinematics, would happen locally for each component. When changes are performed to the parametric model, the simulation must only be re-calculated for the changed components rather than all.

Placing the *Core* component at the downstream end of the process only informs the *Core* component that some or all of the input data has changed, prompting it to recalculate the solution.

However, even if all components knew the relevant robot simulation settings, only uniquely defined positions, such as axis or PTP movements (with given Status and Turn values), would allow for a precise simulation. The – as per Chapter 3.5 – most popular movement components for linear or circular movements do not contain a complete set of kinematic information but retain the posture defined by a previous PTP or axis movement. As such, it is imperative to know their previous movement to simulate their current position accurately. Therefore, an accurate simulation can only reliably occur toward the end of the data flow, where the input information is complete. This data flow also impacts the optimization potential through multi-threaded programming.

Due to this lack of information in the flow-based programming system, the individual movement components are wrapping the relevant function within the `KUKAprcLibrary.PRC_Commands` namespace only performs the calculation from the given plane to XYZABC values (`PRC_CartesianVals`, see Chapter 3.2) as well as the translation from NURBS geometry to polylines in the case of circular and spline movements (see Chapter 4.2).

Without information on the type of robot, the local coordinate system, etc., no accurate robot simulation is possible at the beginning of the data flow.

4.3.2 KUKA|prc Data Input/Output

The main logic of KUKA|prc is designed to work within this set of constraints. The *Core* component in KUKA|prc for Grasshopper essentially wraps the public `KUKAprc` class, whose public parameters largely match the input nodes: A list of commands (`PRC_CommandData`), a settings object (`PRC_GUISettings`), the robot (`PRC_RobotData`) and tool (`PRC_ToolData`) definition, a list of meshes as collision geometry and a numeric value to set the simulation position as a normalized value between 0.0 and 1.0.

The `HostCustomization` property can be set to override the entire `PRC_HostCustimizationMethods` class. Currently, only the collision checking may be overridden by the native functionality of the host software.

To use KUKA|prc, the class `KUKAprc` must be instantiated, and the individual public properties must be set. Calling the `RunPRC()` method prompts the kinematic simulation to run and creates a `PRC_ReturnData` object, which can also be retrieved through a public property of the KUKAprc class.

To get a new simulation output, it is always required to call the `RunPRC()` method. This would mean that the entire toolpath must be recalculated even if only the simulation slider is moved to change the currently simulated position. Obviously, this is not efficient when all other simulation values remain unchanged.

KUKA|prc's approach to identifying new and old data through a hashing algorithm creates a unique integer for each component. To do so, the `GetHashCode()` function is performed on the result of each object's `ToString()` method, creating an `Int32`. Microsoft warns that the `GetHashCode()` function should not be used for any permanent applications, e.g., as a key to retrieve an object from a dictionary; however, within the scope of KUKA|prc, the hashes are only used temporarily, and it does not matter if they change between different versions of the .NET Framework.

Except for the simulation slider floating point value, hashes are used for all input classes, i.e., commands, settings, robots, tools, and collision geometry, because changing any of these values would require a complete recalculation of the simulation process.

While the `PRC_Library` namespace provides easy-to-use static methods for generating commands, tools, and robots, the `PRC_GUISettings` are expected to be defined by an external user interface, such as the one implemented in WPF for use in Windows environments (see Chapter 3.5).

Internally, simulation results are buffered to react rapidly to new slider input. The resulting `PRC_ReturnData` object contains several public properties (Figure 22):

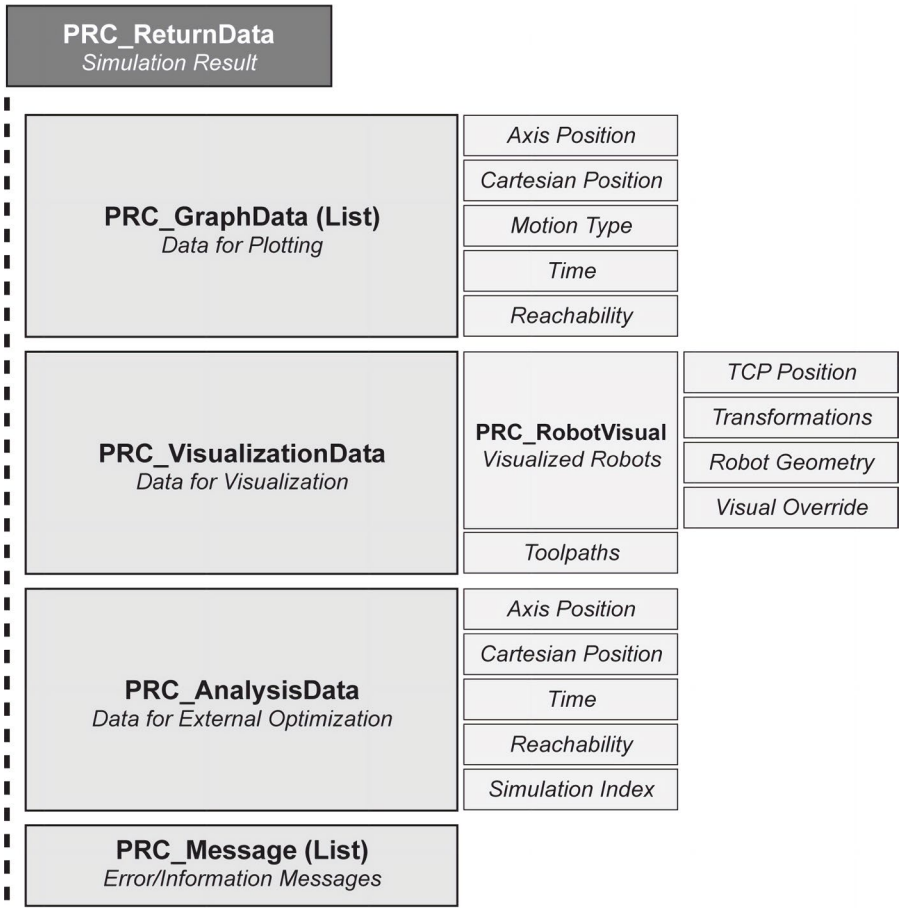

Figure 22: Data returned from the KUKA|prc robot simulation.

The `GraphData` property lists `PRC_GUIData` objects generated for each simulated position. It contains the robot's axis position, Cartesian position, motion type, simulated time, programmed speed, occurrence of collisions, and singularities. This data is mainly meant for purposes such as the Analysis graph implemented in the Windows GUI or possibly automated optimization algorithms.

The `VisualizationData` property is a single `PRC_VisualizationData` object containing the data required to

visualize the robot in a 3D environment. It includes several lists of polylines, defining the toolpath, the coordinate systems for each programmed position, and a polyline representation of the robot's kinematic chain. Its most important property is the list of `PRC_RobotVisual` objects, each defining a simulated robot. Suppose a robot has seven "joints" (base, A1-A6). In that case, the object contains seven `PRC_MeshCollection` objects, defining the robot's geometry, and a list of seven coordinate systems that define each joint's transformation from its base position to its calculated position.

This transformation is always absolute and unrelated to the previous movement. In addition, the `ovColors` property defines override colors, to be used when, e.g., a joint exceeds its defined range or collides with the environment. Further properties include the coordinate system of the robot's tool (`toolPln`), the definition of the robot's cell as a `PRC_MeshCollection` and a dictionary for setting application-specific data.

The `AnalysisData` property contains data that is partly identical to the `GraphData` but formatted as generic value types so that they can easily be output and postprocessed. Most properties are nested lists, providing multiple data points for each simulated position. Data includes the axis values (ten `double` values, six for internal, four for external axes), the occurrence of collisions per axis (ten values), the reachability per axis (ten values), the transformation per joint as a `Plane` (up to ten values), the state of the robot's inputs/outputs (one `string` for each used input/output). Further values provide the calculated time for each simulated position, the index of the currently visualized position, and the resulting KRL code as a single `string` object.

The `Log` property contains a list of error messages and notifications, each consisting of a `string` and an `enum`, defining the message's severity.

Finally, the `Robot` property contains the definition of the robot. The two Boolean properties, RanToCompletion and FullRefresh, declare whether `RunPRC()` finished without errors and if a complete refresh of the visualization is required, which would, e.g., necessitate another conversion from KUKA|prc's internal mesh format to the mesh format used by the host software.

4.4 Core Calculation Walkthrough

The core goal of KUKA|prc is to provide an accurate, responsive robot simulation and code generation. Its main functionality is contained within the `RunPRC()` method, which takes a list of `PRC_CommandData` containing the various movement commands, a `PRC_RobotData` and `PRC_ToolData` for setting the tool and robot, as well as a list of meshes as collision geometry and a normalized `double` value setting the relative position within the simulation (see Chapter 4.3). The data flow is sketched in Figure 23.

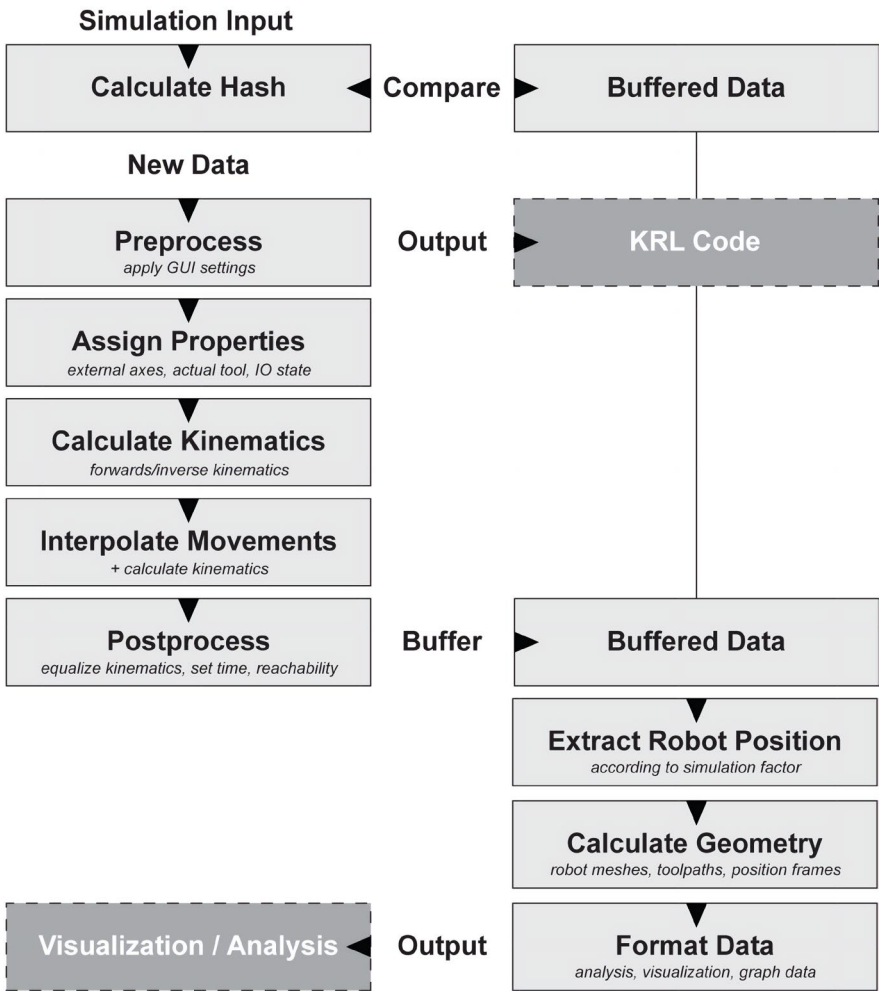

Figure 23: KUKA|prc process visualization.

A major challenge in robot simulation is that robot positions are often not uniquely defined due to the complexity of the kinematics. For these types of movement, it is necessary to know not only the programmed position but also the previous position. While it is perfectly feasible to iterate through a list of commands, this sequential approach makes it hard to use the multithreading capabilities of modern CPUs and programming languages.

Another challenge lies in interacting with the simulation through the normalized simulation value. By default, the simulation resolution is set to 10mm: If two positions are further apart than 10mm, robot positions are interpolated in-between, a process that is particularly important to spot singularities on long, linear toolpaths as well as the non-linear movement towards a point-to-point position. When interacting with the simulation value, e.g., through a slider in Grasshopper, users want finer-grained control than the simulation resolution offers. At the same time, it is not desirable to completely re-compute the entire toolpath, as this would significantly impact the user experience with long computation times.

As a first step, the hash values of each motion command, as well as the robot, tool, and collision data, are combined into a single value. This value is now compared against a buffered value. If they match, the algorithm skips to just simulating the newly defined position along an already simulated toolpath. If they do not match, the algorithm starts the complete kinematic calculation.

4.4.1 Preprocessing Data

The calculation starts by creating a deep copy of all commands to prevent inheritance issues with downstream data. For example, a `Plane` object representing the robot's tool frame may be transformed when the simulation includes an external rotary axis. Whenever the flow-based programming environment requires a recalculation, that transformation would be applied again on the previously transformed geometry, creating errors in the simulation.

Once copied, an incremental ID is assigned as a `double` to each component. The code now performs a binary search to identify the first

movement component within the list of commands. While a toolpath usually starts with a motion command, the user may want to toggle IOs before moving the robot. Beginning with a non-ambiguous motion is important to ensure a reliable simulation. If the first motion is, e.g., a linear, circular, or spline movement, its first position is extracted and used to insert a PTP movement with the same coordinate and a status value as defined in the settings. No special action is taken if the first motion is a PTP or axis movement. An axis position, defined through the "Start Position" field in the KUKA|prc settings, is inserted into the list as the first movement. At the same time, the end position may be optionally omitted.

At this stage, KUKA|prc has to check whether any external axes are attached to the robot model. Currently, KUKA|prc supports up to four external axes (for a total of 10 axes), which can be either rotary, linear, or "dummy" axes that do not have any effect on the simulation, used, e.g., for driving extruders through KUKA robots.

Geometrically, the difference between rotary and linear axes is clear, as one rotates and the other translates. Concerning the KUKA|prc simulation, KUKA|prc assumes that every external axis uses a so-called external base. That signifies that when the value of a rotary axis changes, it rotates the previously defined external base coordinate system.

Therefore, the programmed position within that coordinate system stays the same; however, in relation to the robot's world coordinate system, the programmed position is rotated around the axis accordingly. In contrast, a linear axis does not move the programmed position. Instead, it moves the robot itself, essentially changing its base coordinate system within the given global coordinate system.

These transformations can be put into a hierarchy, with a two-axis positioner like the KUKA DKP series first tilting the positions via one axis and then rotating them via another. The same applies to linear axes, which can be combined to create, e.g., a three-axis portal setup.

KUKA|prc provides three ways of setting the axis position of an external axis. By default, they are set to `double.NaN`, interpreted as just keeping the previously defined value. If no values for external axes are specified in the code, the initial axis movement always sets a value for E1-E4.

Alternatively, KUKA|prc can be set to calculate the external axes' position automatically.

For linear axes, the user needs to define a minimal, ideal, and maximum distance between the robot's position (its ROBROOT) and the programmed position. Geometrically, this can be considered an intersection between a ray and a sphere, with the sphere being the ideal value. This intersection results in either zero results (in which case the closest point is taken), one result (if the distance between the sphere and the ray is identical to the sphere's radius), or, most commonly, two results.

KUKA|prc then chooses the intersection point closest to the previous position. The robot will stay at that position of the external axis until it exceeds the minimum or maximum value, in which case it will reposition again. This is meant to increase the accuracy, as any movement of the linear axis might reduce the robot's absolute accuracy. The user can set all three values to the same number if a constant repositioning is desired.

The software also provides a solver for the rotary axis, which is expressed as a vector that should set the direction of the tool axis in relation to the global CAD coordinate system. In an ideal case, the algorithm projects both the orientation vector and the current Z-axis of a robot frame into the rotation plane of a rotary axis. Then, it calculates the angle in degrees between them. Various border cases, e.g., having the Z-axis of the robot frame precisely perpendicular to the vector, are also considered.

With the Cartesian position of each movement command fixed, KUKA|prc iterates over the entire list of commands, collecting and buffering sequential values such as speed, chosen tool, state of digital/analog outputs, and kinematic posture. In the case of the tool, the initial tool is assigned to each command unless a Change Tool command changes the current tool, in which case the new tool gets assigned for each subsequent command.

Similarly, as the actual KUKA robot saves the Cartesian and PTP speeds independently in two variables, the previous speed will be assigned unless the motion command assigns a new speed. Finally, Cartesian movements inherit the posture of the most recent PTP movement.

KUKA|prc already solves the forward kinematics of axis commands to calculate the posture of axis movements. As axis movements are unambiguously defined, the forward kinematic solver can be heavily parallelized and benefit heavily from multi-core systems.

4.4.2 Code Generation

There is now sufficient data to create the file output. The generation of the file is implemented as a separate, asynchronous task to ensure that, e.g., delays from accessing a network folder do not negatively impact the responsiveness of the software. KUKA|prc supports several different data formats. The default is KRL. However, it also supports the tech packages KUKA.CNC and KUKA CAMRob, as well as Sunrise (see Chapter 4.7). The general logic, however, is identical, as the algorithm iterates over the individual commands and builds up several lists of `strings`, with the exact syntax depending on the output format. For example, the KRL option generates an SRC file where a linear movement is expressed as in Code 6, line 1.

```
1 LIN {X 75.514, Y 34.74, Z 20, A 0, B 90, C 25.924}
2 G1 X75.514 Y34.74 Z20 A0 B90 C25.924
3 1.00;75.51;34.74;20.00;0.00;90.00;25.92
```

Code 6: The same movement expressed in KRL (line 1), KUKA.CNC G-code (line 2), and as a CSV file for KUKA.CAMRob (line 3).

KUKA.CNC is a tech package that allows the user to import G-code of any length, providing CNC-optimized interpolation strategies and other benefits. Its NC file would express the linear movement as in Code 6, line 2.

Finally, KUKA.CAMRob allows the robot to process large CSV files through a ring buffer without running into memory limits. CAMRob creates a master SRC file that calls sequences of linear movements via the `CR_PROCESS_FILE` KRL command. Its CSV file would express the linear movement as in Code 6, line 3.

The exact order of commands is set through a so-called pattern. KUKA|prc parses the pattern and then replaces, e.g., the line containing [COMMANDS] with the list of commands. This allows the user to

optionally include, e.g., subfunctions operating external tools. A pattern can be integrated into a robot model to account for specific use cases.

4.4.3 Kinematic Simulation

The kinematic simulation takes the same data used in the code generation but discards all non-movement-related commands, as they do not impact the kinematic solution. While it was previously stated that the kinematic calculation of Cartesian movements could not be easily parallelized due to the previously calculated forward kinematics, KUKA|prc is now aware of the posture of each Cartesian movement. This makes it possible to parallelize the process similarly to the forward kinematic calculation.

As the individual calculations are not long-running, KUKA|prc does not create a thread for each position but uses C#'s `Parallel.For` loop to dynamically create tasks and manage them via the `ThreadPool`. While the performance scales with more cores, it is not linear due to the still-existing overhead of creating a task rather than just performing the computations in a simple for-loop. As the order of the results cannot be predicted, KUKA|prc collects the results in a `ConcurrentBag` and then sorts the list according to each command ID.

Positions that are entirely outside the range of the robot cannot be calculated. In this case, only the A1 value is assigned, with all other values set to 0, so the robot points towards the unreachable position. The next step in the simulation is the interpolation of commands. While it can be optionally disabled, performing the interpolation significantly improves the reliability of the simulation. The interpolation process is simple, but due to the number of combinations of different movement types, it still amounts to a large amount of code. KUKA|prc now knows the position of each command, both in terms of axis values and Cartesian position.

Every robot program starts with an axis command; the type of interpolation now depends on the subsequent movement type. If the next movement is a PTP movement, the algorithm takes the previous movement's axis values and the current movement's axis values. It interpolates the axes linearly so that all axes arrive at the position of the

current movement simultaneously. The number of interpolated commands depends on the simulation resolution and the distance between the positions.

A distance of 200mm with a simulation resolution of 10mm would result in 20 interpolated movements. Meanwhile, a distance of 5mm with a simulation resolution of 10mm would not generate a single interpolated movement. The distance is measured along the arc/spline for spline and circular movements. Once the interpolated commands have been generated, the forward kinematics solver calculates the Cartesian position for axis-based interpolation, and the inverse kinematics solver the axis position for Cartesian-based interpolation.

By default, the robot takes the shortest path between two positions. Some robot axes have an axis range of less than 360 degrees (e.g., -190 to 45 degrees in the case of a KUKA Agilus' A2), through which the axis position is uniquely defined. However, other axes (e.g., -350 to 350 degrees for a KUKA Agilus' A6) allow for redundant axis values, e.g., the position of -190 degrees could also be expressed as positive 170 degrees).

Unless the axis position is precisely defined, as is the case for axis movements and PTP movements that provide both status and turn, KUKA|prc iterates through the list of commands and equalizes the axis values accordingly to replicate the actual robot's behavior.

4.4.4 Toolpath Analysis

With that step, the robot's movement is now simulated, and KUKA|prc can extract additional information for visualization and analysis. The first step is to calculate the time value for each robot position. For axis movements, KUKA|prc knows the initial axis position, the target axis position, the maximum axis velocity - as published in the robots' data sheets – and the movement's programmed velocity in percent. It can now determine the duration of each axis' movement and picks the slowest (=highest) time as the timestamp. For Cartesian movements, the calculation divides the path length (mm) by the programmed speed (m/sec), taking care to convert the units accordingly.

A limitation of the simplified approach is that the algorithm does not take acceleration and deceleration under consideration, as the required values are not provided by the KUKA specifications. As such, toolpaths with many changes in direction may end up with significantly longer process times than calculated. The time stamp, therefore, acts as a best-case calculation that can still be used as a relative value for, e.g., optimization through evolutionary solvers.

The timestamp calculation is also influenced by the specific RoboTeam commands, which can synchronize two or more robots. KUKA|prc allows users to define positions where the robots wait for each other (`#ProgSync`) or move synchronously (`#MotionSync`). Doing so is achieved by adjusting the time values of both robots, adding simulation-internal wait commands for `#ProgSync` when needed, or slowing down the speed of movements to the slowest robot's speed, similar to axis movements. The data exchange is achieved through a static `ConcurrentDictionary`, where the robot's RoboTeam ID is used as the key. As such, one robot needs to be set up as robot 1, with 2 as partner and the other robot as 2 with 1 as partner.

As the next step, KUKA|prc tags unreachable positions, differentiating between axes out of their designated axis range and geometrically unreachable positions. This is represented as a `PRC_Axvals` object – containing ten axis values as `double` and an index as `int` - with an axis either being assigned a 1 for collision or a 0 for no collisions. The `index_value` parameter is set to 1 if the object contains any unreachable position. In the case of a position that is geometrically impossible to reach, all values are set to 0.

At this stage, KUKA|prc checks for kinematic singularities, which most commonly result in infinite – or very high – velocity at an axis, stopping the robot. By knowing the time of a movement and the difference in axis values, the algorithm can determine the speed for each axis. Suppose the speed exceeds the maximum speed of an axis as defined in the specifications. In that case, this position is marked to warn the user of possible singularities where such behavior would occur. (Figure 24)

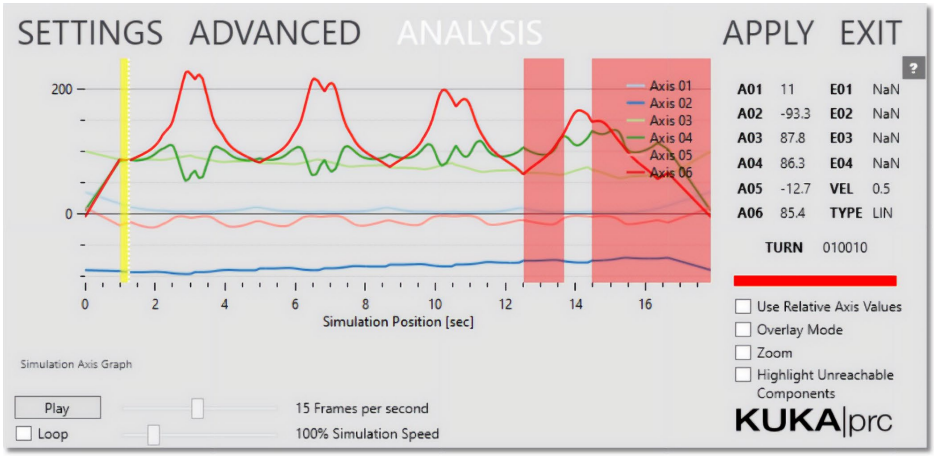

Figure 24: Visualization of toolpath analysis for excessive speed/singularities (yellow) and unreachable positions (red).

As the final, optional analysis check, KUKA|prc can perform collision checking (see Chapter 4.2). KUKA|prc assumes that a robot cannot collide with itself and, therefore, checks for collisions between the tool and the robot joints and between the optionally set collision geometry and robot joints. Regarding the simulation, the optional robot cell geometry is considered part of the robot's base geometry.

This completes the part of the calculation that is only called if the current hash differs from the buffered hash. Now, the current hash is assigned to the buffer, so identical calculations are not repeated.

4.4.5 Simulation Update

With the entire toolpath known, the cyclically called section of `RunPRC()` calculates the displayed position as set by the normalized simulation value and formats the data to be output.

To determine the robot position to simulate, KUKA|prc takes the timestamp of the last motion command and multiplies it with the normalized simulation slider value. A binary search function now searches for the closest timestamp to the calculated time. If the time does not match precisely a timestamp, KUKA|prc takes the previous and next timestamp and expresses the simulated time as a normalized

factor. With a prior timestamp of 17 and a next timestamp of 19, the simulation value of 18 would be at a factor of 0.5.

This value is now used to interpolate between the two robotic movements, using the same logic as for the general toolpath interpolation. While the exact position is calculated – depending on the involved movement types - via inverse/forward kinematics, collisions, singularities, and output states are not recalculated but taken from the movement closest to the simulated position, according to its interpolation factor. To visually differentiate between parts of the toolpath that have already been traced and parts that have not, the factor is also used to split the toolpath accordingly so that a different color may be assigned during visualization.

The data for plotting the axis movement, collisions, and singularities on the GUI's Analysis graph is formatted as a list of `PRC_GUIData` objects. In contrast, the data for the component's analysis output (axis values, Cartesian values, collisions, etc.) is collected into a single `PRC_AnalysisData` object without requiring specific postprocessing steps.

For the 3D visualization, KUKA|prc now creates a `PRC_VisualizationData` object. The toolpaths are included as a list of polylines, differentiating via colors between the parts of a toolpath before and after the simulated position and unreachable toolpath positions. A simplified polyline represents the robot's kinematic chain and a series of `PRC_Frame` objects, the current tool coordinate system, and – optionally – the frame of each programmed robot position. Each `PRC_Frame` contains three lines in red, green, and blue representing the coordinate axes and a scaling factor to set their size.

To display the robot in 3D at the interpolated position set by the simulation value, KUKA|prc uses the axis values to sequentially transform the meshes, similar to the forward kinematics solver. To save computational time and allow the host software to use its mesh format optionally, only the mesh transformation matrix is transformed, not the individual vertices making up the mesh. Transforming the vertices requires the user to use the meshes' `Bake()` method.

Finally, KUKA|prc generates a list of `PRC_Message` objects to communicate errors or warnings to the user, e.g., if the save folder does not exist or the toolpath contains unreachable positions. The complete data object containing the accordingly formatted data is now returned as a `PRC_ReturnData` object by the `RunPRC()` method.

4.5 Challenges in Multi-Platform Integration

The following sections showcase four different environments within which KUKA|prc can operate:

- A console application representing the conventional programming approach, where one defines a problem through code, runs an algorithm, and then receives a result. The advantage is that it can be universally applied, from a command-line interface to within another C# application (see Chapter 5.4).
- Visual programming through either Grasshopper or Dynamo provides a much higher degree of abstraction to the user but, in exchange, requires the developer to work within a range of constraints that dictate how data is collected, processed, and output. As such, the integration is much more complex and accessible to non-expert users.
- Finally, game engines like Unity provide a particular challenge as they are not updated once (console application) or whenever data is changed (visual programming) but constantly. This behavior also leads to different expectations from the user: For a console application, it might not be immediately apparent if the calculations take a second more or less, while visual programming starts feeling unresponsive with 150 milliseconds or more. A similar delay within a game engine would lead to a highly negative user experience if the screen freezes while KUKA|prc performs some calculations.

While KUKA|prc was intended and optimized for visual programming, the exemplary integrations as a console application or a real-time algorithm running within a game engine show that the developed

software is sufficiently flexible and can be adapted very closely to the users' very different needs.

4.5.1 Exemplary Console Application

While a console application is not specific to the challenges of working with visual programming, it showcases the general data flow and the developed programming approaches for KUKA|prc. The emphasis here is on the high degree of abstraction that enables programmers with basic robot experience to use KUKA|prc rapidly (Code 7).

```
1 KUKAprc Core = new KUKAprc();
2 List<PRC_CommandData> cmds = new
List<PRC_CommandData>();
3
cmds.Add(KUKAprcLibrary.PRC_Commands.PRC_PTPMove(new
Plane(new Point3d(600, 0, 200), new Vector3d(1, 0,
0), new Vector3d(0, 1, 0)), 10, "010", "C_PTP"));
4
cmds.Add(KUKAprcLibrary.PRC_Commands.PRC_LINMove(new
Plane(new Point3d(600, 500, 200), new Vector3d(1, 0,
0), new Vector3d(0, 1, 0)), 0.25, "C_DIS"));
5 Core.InputPRCCommandData = cmds;
6 Core.InputSimulationSlider = sliderval;
7 Core.InputPRCSettings = new PRC_GUISettings(true);
8 Core.InputRobot =
KUKAprcLibrary.PRC_Robots.PRC_KR610R1100();
9 Core.InputTool = new PRC_ToolData("Toolname", new
PRC_CartesianVals(0, 0, 100, 0, 0, 0, 6), 6,
toolmeshes);
10 Core.RunPRC();
11 Core.OutputData.RanToCompletion ?
Console.WriteLine("KUKA|prc ran to completion");
```

Code 7: Exemplary C# code for a console application.

In Code 7, line 1 instantiates the `KUKAprc` main class. Then, a list of `PRC_CommandData` objects is created and populated with robot commands using the `KUKAprcLibrary` namespace (lines 2-4). The

parameters of the Core instance are assigned in lines 5-9. For the simulation position, the `double sliderval` is used. The `PRC_GUISettings(true)` constructor populates the simulation settings with the default settings while the robot data is taken from the KUKA|prc library. The tool is constructed by setting its XYZABC values, tool number, and a list of meshes.

By line 10, sufficient data is now available to run the simulation and access the resulting `OutputData` in line 11.

While a console application does not benefit from the intuitive approach of visual programming and the plethora of extra functionality and plugins of environments such as Grasshopper and Autodesk Dynamo, it allows the user to either approach problems in a more integrated way, e.g., as a standalone application, or to implement it into custom programming strategies, through e.g. the C# or the Python component, which can access C# libraries via IronPython (A. Harris 2010).

4.5.2 Grasshopper Integration

Embedding a library into a visual programming environment like Grasshopper requires many customizations and data translations. In KUKA|prc, the relevant functions are primarily provided as static methods in the `KUKAprcLibrary` namespace, with several mandatory and optional parameters. However, these methods cannot immediately be used with Grasshopper, as despite identical naming conventions (e.g., `KUKAprcCore.Geometry.Point3d` and `Rhino.Geometry.Point3d`), the data types cannot be directly interchanged. More importantly, Grasshopper requires all components to derive from the `GH_Component` class.

As such, for each component, a class deriving from `GH_Component` needs to be created, setting the icons and description texts, defining a unique GUID to identify the component (see also Chapter 3.5), and creating the appropriate input and output parameters, providing a data type and description. Proper labeling, descriptive texts, and clear icons are crucial for making software accessible to users by enabling self-learning and guided experimentation.

The actual computational work happens within the `SolveInstance` method, which first collects the data from the various inputs and assigns them as references to previously created objects. As such, geometries need to be converted from Grasshopper's `Rhino.Geometry` model to KUKA|prc's custom geometric framework (see Chapter 4.2). Then, the geometry can be passed as a parameter to the relevant methods in the `KUKAprcLibrary` namespace.

Returning data requires a careful look into the data types and structure. Output nodes of Grasshopper components can either contain a single item, a list of items, or a tree of items. While single items and lists do not require special treatment, no generic data tree-like collection in C# exists. Therefore, Grasshopper uses its `DataTree<T>` to represent trees; e.g., nested lists of elements must be converted to that format.

Grasshopper outputs all data as classes deriving from either the abstract class `GH_Goo<T>` for generic data or `GH_GeometricGoo<T>` for geometric data. Standard Rhinoceros 3D geometry such as a `Rhino.Geometry.Mesh` is automatically cast to `Grasshopper.Kernel.Types.GH_GeometricGoo<Rhino.Geometry.Mesh>`. These so-called params contain not only the geometry itself but also an icon representing that type of data, tooltips, serialization instructions, and a `ToString()` method. Therefore, the Grasshopper implementation of KUKA|prc defines five different params: `CommandData`, `AnalysisData`, `RobotData`, and `ToolData`.

Another critical aspect of usability is the seamless integration of the user interface. KUKA|prc's GUI for Windows is contained in the `KUKAprcLibrary.Win.Gui` namespace does not include any specific Grasshopper data so that it might be used from any Windows software, even the console application. An advantage of Grasshopper is that even after the `SolveInstance` method has concluded, the instance of the `GH_Component` and its properties are kept intact.

Therefore, as long as the instance of the `KUKAprc_GUI` is declared within the class itself, it is possible to have multiple GUIs on screen, one for each *Core* component, that can interact with its *Core* component without having to wait for a recompute of the Grasshopper definition.

While the *Core* component can directly set the data displayed in the GUI via a simple, thread-safe method, the communication from the GUI to the *Core* component is event-based, with the *Core* component subscribing to the GUI's `EventHandler`. Two different events can be communicated – when the simulation position is changed in the Analysis view and when any GUI settings are changed.

Either way, the event triggers a method that waits until the Grasshopper component is no longer busy by querying the `SolutionState` of the `GH_Document` and then prompts the component to refresh. The *Core* component then computes a new solution based on the changed data.

The final step is the visualization in the 3D viewport (Figure 25). The `VisualizationData` property of the simulation result contains the relevant meshes and polylines that can be output directly by the component. However, Grasshopper's mesh output does not provide different colors for either polylines or meshes, which has both an aesthetic and a functional impact, e.g., it is no longer immediately possible to see which robot joint is out of reach or colliding with its environment.

This can only be achieved by drawing directly to the viewport rather than outputting the geometry into Grasshopper. To do so, it is necessary to override Rhinoceros 3D's `DrawViewportMeshes` and `DrawViewportWires` methods and enable them to access a list of `GH_CustomPreviewItem` in a thread-safe way.

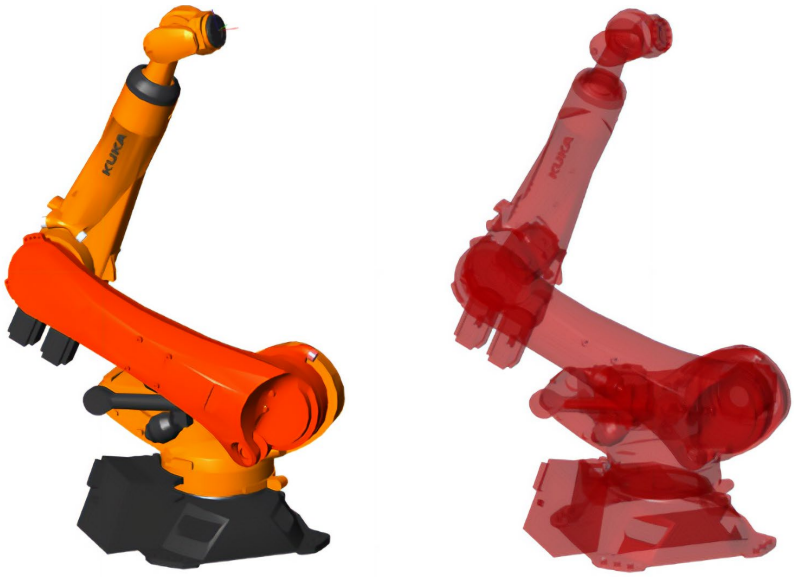

Figure 25: KUKA|prc display pipeline with colored mesh showing the unreachable position (left), Grasshopper default mesh display mode (right).

These methods are called whenever the viewport is changed; as such, it is essential to keep them as performant as possible. Due to that, a buffering mechanism is created that translates the un-transformed robot meshes from KUKA|prc's format into a native `Rhino.Geometry.Mesh` object. Whenever the simulation updates, the algorithm checks whether the buffered robot geometry needs to be updated, e.g., when the robot model is changed. Then, it transforms the native Rhino geometry, casts it as a `GH_Mesh`, and sets it as the `m_obj` property of a `GH_CustomPreviewItem`.

The exact process is used to draw the polylines, as a complex toolpath may consist of hundreds of thousands of points, offering a significant performance increase through buffering. When transparent geometry, e.g., for visualizing robot cells, is used, the drawing order must be considered.

Integrating KUKA|prc into Grasshopper shows that a significant effort and understanding of the host software is required to achieve a native feel, even if the new program was explicitly developed with that host software in mind.

4.5.3 Dynamo Integration

Autodesk Dynamo (Keough 2020) is built on a paradigm similar to Grasshopper, as its original developer aimed to bring Grasshopper to the Building Information Modeling (BIM) software Autodesk Revit. In version 1.3, Dynamo shows how slightly different approaches in the host software can require significantly different data flows for a plugin.

In general, Dynamo's so-called Zero Touch interface significantly facilitates the plugin development, as it turns every library public method into its node. While this is a very valuable approach for prototypically implementing new algorithms, it leads to several challenges on the UI side.

In the case of KUKA|prc, the `KUKAprcLibrary` namespace would be a very fitting use-case for Zero Touch, as it consists of a limited number of methods that are grouped appropriately, e.g., with the `KUKAprcLibrary.PRC_Robots` class containing the `PRC_KR22R1610()` method that creates a KUKA KR22R1610 robot. The core library is less valuable to the end-user as it includes many more methods and events that are usually not exposed to the user of a visual programming environment.

A significant architectural challenge of Dynamo is the division between a component's interface and actual logic as per its MVVM pattern. Where Grasshopper can have both within a `GH_Component` class, making the sharing of data easy, Dynamo requires a component deriving from its `NodeModel` class that defines the inputs and outputs, description, serialization, etc., and uses the `AstFactory` class to create a `System.Func<T>` delegate out of a method, passing execution to another static function that has to be located in a separate assembly. For example, the `PRC_ChangeTool` method that takes a tool definition as the input and outputs a command to switch the tool would be implemented as in Code 8. The result of the function call is then assigned to the node's first (0) output.

```
1 var functionCall =
2     AstFactory.BuildFunctionCall(
3     new Func<PRC_ToolData,
PRC_CommandData>(Logic.PRC_ChangeTool),
4     new List<AssociativeNode> {tool});
5
6 return new[] {
AstFactory.BuildAssignment(GetAstIdentifierForOutputI
ndex(0), functionCall) };
```

Code 8: Division between a component's interface and logic.

Compared to Grasshopper, this approach has the drawback that it cannot easily keep data persistent between calls to the node, complicating data buffering. In the first release of KUKA|prc for Dynamo, the GUI was therefore integrated as a static object, limiting the user to a single instance of the KUKA|prc *Core* component. The current approach creates a dictionary for each persistent data object, e.g., `ConcurrentDictionary<string, KUKAprc_GUI>`, where the `string` refers to the unique GUID of the node. Through that approach, the relevant instance of the GUI can be retrieved from the dictionary.

The general interaction with the GUI is easy because Dynamo natively uses WPF to draw its components. The GUI is, therefore, displayed like any other WPF window/control.

Unlike Grasshopper, which uses the viewport of the host software Rhinoceros 3D, Dynamo displays its geometry within its viewport, using WPF's 3D engine exposed through the open-source Helix Toolkit. Like Grasshopper, the geometry output by Dynamo nodes does not contain color information; as such, it is necessary to draw the geometry directly into the Helix viewport.

To do so, the geometric information for both meshes and polylines is combined into an `IRenderPackage`, which in turn is added to the `BackgroundPreviewViewModel` – retrievable through the known GUID of the KUKA|prc *Core* component - via the `AddGeometryForRenderPackages` method and thus drawn in the viewport.

From the user experience, the KUKA|prc integrations into Grasshopper and Dynamo are mostly identical, even though the underlying logic is very different. While Dynamo's MVVM implementation is architecturally more modern than Grasshopper's approach, it necessitates specific workarounds to achieve the same goals.

4.5.4 Unity Integration

Grasshopper and Dynamo are closely related software within the scope of visual programming. One of the core concepts of visual programming is that changes propagate through the graph, which works well for intuitive interaction with code but causes issues when confronted with real-time data (see Chapter 4.6). However, Unity is a general 3D engine focused on computer graphics and real-time interaction. This leads to two specific challenges: Different coordinate systems and constantly updating data.

While Grasshopper and Dynamo both use right-handed coordinate systems where Z+ is facing upwards, Unity follows the left-handed coordinate system, more commonly used in computer graphics, where Z+ is instead the depth of the rendered images. KUKA|prc internally uses the right-handed coordinate system, so to keep everything consistent, it is necessary to transform geometric data first to KUKA|prc's format and then back again. To transform a point object from Unity's to KUKA|prc's coordinate system, its Y-value is switched with the Z-value. For transformations, the rows and columns of an object's matrix need to be swapped (row-major versus column-major).

However, as Unity does not allow objects to be transformed directly through a matrix, the matrices must be split into a translation and an orientation (quaternion) component. The translation value is easily extracted from the `Matrix4x4`, and the quaternion can be determined through Unity's `Quaternion.LookRotation (forward, upwards)` method, deriving the two vectors from the rows of the matrix.

Loosely connected to the coordinate system is Unity's unit system. A KUKA robot – and therefore also KUKA|prc – works with millimeters as units. However, while Unity does not have a default unit system per se, its physics system and values relating to drawing distance, etc., are set

up for meters as units. Changing all simulation values is more complicated than simply dividing all calculated Cartesian robot coordinates by 1000 before they are displayed and multiplying all input geometry coordinates by 1000.

The second challenge is the game-focused approach of having algorithms constantly display new results at the game's cycle time. KUKA|prc's Unity integration is implemented as a class deriving from `MonoBehaviour`, consisting of a `Start()` and an `Update()` method by default. While the `Start()` method is only called once at the beginning, the `Update()` method is called at the application's cycle time. A similar approach is, e.g. used in physical computing in the Arduino IDE. That means that if an application runs at 60 frames per second, Unity will call the method 60 times per second. If the method takes too long (17ms if there are no other tasks to achieve 60 frames per second), the game's cycle time will drop, leading to a stuttering display.

In such an example, an efficient buffering process implemented by KUKA|prc is essential, ensuring that only the most relevant geometries are recalculated. Calculating a single position per the simulation slider input within 17ms is not a problem with a reasonably powerful PC. However, generating a toolpath consisting of more than a thousand positions will exceed the available time, requiring a different approach.

A current implementation of KUKA|prc calculates new data asynchronously and displays it when ready. While doing so would cause problems within the scope of visual programming, it is a perfectly suitable approach for a game engine that runs with a constant background refresh.

4.6 Realtime Interaction

With the increasing number of processes that, e.g., implement mass customization or depend on the computational capabilities of the cloud, there is a rising demand for real-time control of robotic arms through external PCs or microcontrollers.

4.6.1 Interfaces for Realtime Control

KUKA offers application-agnostic communication interfaces, such as the RS232 serial port for older pre-KRC4 controllers, KUKA Ethernet.KRL for TCP and UDP communication via Ethernet and many industrial field buses such as EtherCAT, ProfiNet, and Ethernet/IP. These interfaces have in common that they deal with generic data: The serial port's data can be interpreted as either binary or ASCII data, and the binary data coming from field buses can be individually assigned to data types through the KUKA WorkVisual environment, e.g., interpreting 16bit binary data as a `UINT` unsigned integer.

KUKA Ethernet.KRL adds additional functionality by working with a custom XML structure: This data structure itself is defined as an XML file, setting the data types of each field as well as parameters regarding the data flow (IP address, port, protocol, buffer, etc.). The robot can then receive and interpret the data cyclically (through the Submit interpreter) or at a specific position within the code and reply with its XML-formatted datagrams.

These protocols are ideally suited to sending single, non-blended movement commands and other information. However, it is a significantly more considerable effort to create a KRL program that works like a ring buffer that receives data, puts it into a queue, and then executes the movement commands whenever the advance pointer requires new data. Anything beyond a very limited scope would also require additional functionality for setting tool and base, jogging the robot, canceling already submitted movements, etc.

Another approach that already considers robot movement is the KUKA Realtime Sensor Interface (RSI, see Chapter 5.1). RSI is meant for processing sensor data in real-time and can receive data from either field buses or via Ethernet, using an XML-based communication setup strategy similar to Ethernet.KRL. The sensor data is most commonly used to offset robot movement in hard real-time.

For example, a robot would follow a toolpath for polishing, programmed as a series of linear movements. A force sensor provides the actual vertical force along Z, causing the robot to offset its path until the desired force value is achieved without changing the base programming. The

necessary transformation, offsetting, and filtering of the data can be defined via WorkVisual in a simple visual programming environment (Figure 26). While the corrections are often in the range of a few millimeters, they can be set to go beyond the robot's workspace, allowing real-time control of the robot.

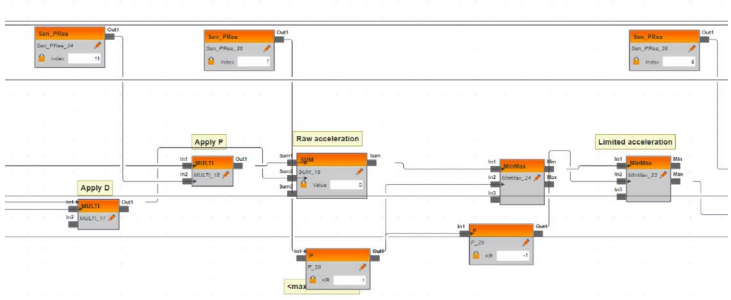

Figure 26: Visual programming for real-time sensor interaction in WorkVisual.

The challenge of using RSI is that the robot expects a correction value for every robot cycle, i.e., every 4-12 milliseconds. Such a cycle time can only be achieved reliably with deterministic environments such as industrial PLCs. A typical PC-based operating system like Windows or a non-real-time Unix system would be able to reply within the cycle time most of the time but would be delayed at least a few times by, e.g., drivers or other services.

This can also be observed in audio production (Brandt and Dannenberg 1998), where latency spikes are immediately hearable and, therefore, a big concern. In the case of a real-time controlled robot, a latency spike would cause the machine to stutter visibly, putting a lot of strain on its gears and possibly leading to an emergency stop to prevent further damage.

On the control side, RSI expects a correction value in either degrees or millimeters. This means that the user does not define the target frame, speed, and interpolation strategy to get there but has to deal with all aspects of motion planning himself. As the cycle time/time step is always identical, a higher correction value leads to a higher speed, while a smaller correction value leads to a lower speed. At the same time, the acceleration and deceleration ramps need to be considered. As such, using RSI for global real-time control is feasible (see Chapter 5.1) but

poses very specific requirements that exclude most visual programming environments.

4.6.2 mxAutomation for KUKA Robots

Compared to the other communication interfaces, all of which were already available for controllers running KRC2, mxAutomation is a more recent interface that was first internally presented in 2013 and for which the Association for Robots in Architecture became a development partner around 2015 (Braumann and Brell-Cokcan 2015).

mxAutomation's head architect, Heinrich Munz, explains the concept of mxAutomation as a "driver" similar to Apple's CarPlay protocol, which defines a communication interface and provides a default library, which then needs to be integrated by the automotive companies to interact with the actual phone.

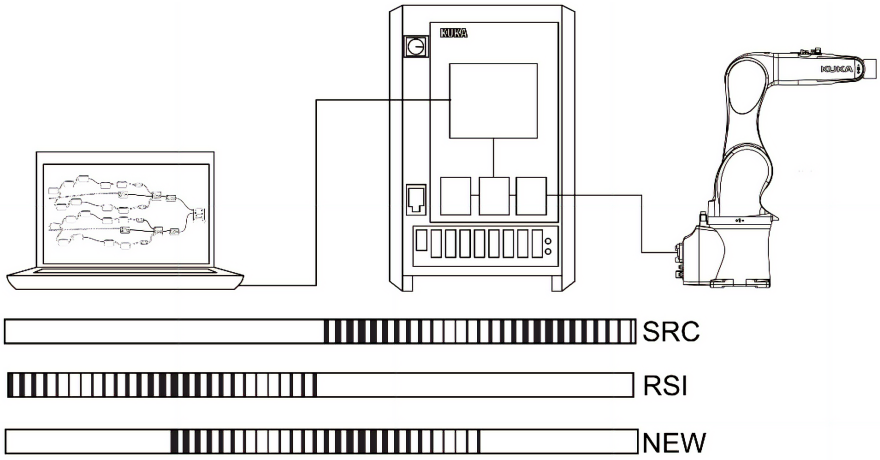

Figure 27: SRC files run directly on the robot, while RSI keeps the entire control logic on the external PC. mxAutomation (labeled "NEW") provides a predefined way of interfacing with a robot in real-time.

In practice, mxAutomation was developed to allow external machines – PLCs, in particular, CNC controllers – to control external robots for, e.g., loading without requiring the user to interact with the robot's controller or to implement highly robot-specific functionality. KUKA, therefore,

provides a library for different PLCs (Siemens S7, Rockwell, CodeSys, Beckhoff) that implements function blocks for robot movement in a PLCOpen-certified layout, enabling PLC programmers to control robots the same way that they control any other machine.

This library structures the data in the KUKA-prescribed format and assigns it to 256 bytes of cyclic process data (256 bytes in, 256 bytes out). It can then be sent to the KUKA robot running mxAutomation through various protocols, from field buses like Profinet, Ethernet/IP, and EtherCAT to raw UDP communication. A ProConOS softPLC running on the robot controller processes the data and communicates with the mxAutomation software.

Received commands are put into a buffer and are processed sequentially following the FIFO logic (KUKA 2017). However, commands can be configured to abort the current movement, flush the buffer, and be processed immediately. As long as the buffer contains at least two commands, the movements can be blended identically to regular KRL. Therefore, while mxAutomation preferably communicates with a real-time PLC, its timing is not millisecond-critical as the buffer will compensate for any short delays, making it suitable to be used by non-real-time systems as well.

4.6.3 PLC-based Dataflow in KUKA|prc

In most geometry-oriented visual programming environments like Grasshopper and Dynamo, nodes are not refreshed continuously but only when upstream changes occur. In contrast, PLCs call their tasks at every cycle. A PLCOpen function block MC_MoveAbsolute (PLCopen 2018) (Figure 28) is equivalent to a robot's PTP movement, though mxAutomation replaces several key data inputs.

Rather than defining the position of a single axis as a REAL, KUKA requires an E6POS structure – similar to KRL - to define a Cartesian position, sets the velocity not as a `REAL` but as an `INT`, provides more complex blending parameters (`APO`) and does not implement the Jerk input.

The parameters relating to the data flow are identical: To execute a function block, the Boolean `Execute` input needs to be set to `TRUE`. In the next cycle, the function block's `Busy` output switches from `FALSE` to `TRUE`. `Active` becomes `TRUE` once the robot performs a movement until finally, `Done` signifies that the movement has concluded. This behavior is used to chain function blocks together rather than calling all function blocks of a potentially complex toolpath every cycle.

By connecting the `Done` output of one function block with the `Execute` input of the next, the following function block is only activated once the previous block has finished processing. If the movements are supposed to be blended, the `Busy` output may be used instead.

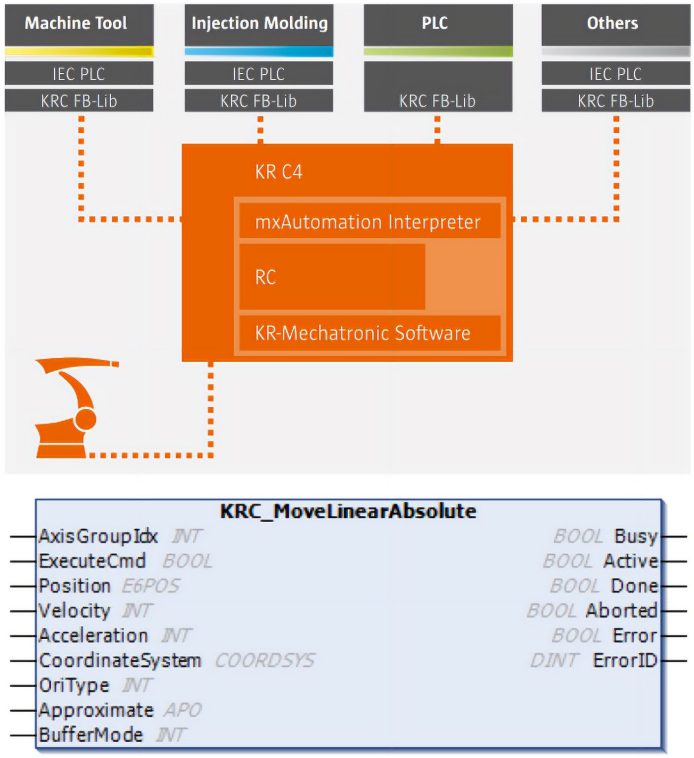

Figure 28: mxAutomation data flow diagram (above, KUKA Robotics) and exemplary PLCOpen function block for a linear robot movement (below).

The C# and C++ code of KUKA's UDP-based mxAutomation library is a direct translation of the IEC 61131-3 Structured Text logic used for the

PLC communication, with each function block directly translated into a C# class. Consequently, to use mxAutomation within KUKA|prc, it is necessary to create a logic duplicating a PLC-based dataflow in C#. By design, KUKA only provides the logic for formatting the data but does not define the socket implementation and general dataflow logic, leaving that up to the library's implementor.

Similar to the real-time integration for Sunrise (Chapter 4.7), command sets coming from, e.g., Grasshopper or any other host software are enqueued at a `ConcurrentQueue<PRC_mxA_BaseData>`, where each `PRC_mxA_BaseData` object contains a list of robot commands as `PRC_mxA_Command`, the current settings as `PRC_mxA_Settings` and a hash value as an `int64`. The `PRC_mxA_Command` class implements the mxAutomation classes defined by KUKA's library – e.g., `KRC_MOVELINEARABSOLUTE` for linear movements - and additional data like the commands Cartesian position, as calculated by KUKA|prc's kinematic solver.

The mxAutomation client in KUKA|prc is contained within the `KUKAprc_mxA` class. Calling the `PRC_ControlmxA` method starts or updates a loop containing a series of asynchronous tasks. First, depending on the interaction strategy (see below), the client either dequeues a single command set from the queue or dequeues all command sets and only keeps the most recent one.

While a PLC task is running at a deterministic cycle time, the mxAutomation client's non-real-time cycle is tied to the connected robot: Once data (256 Byte) is received, the code determines the response as quickly as possible and sends it back to the robot (256 Byte), which then replies again. By doing so, the non-real-time system does not need to be aware of the cycle time but provides a best-effort.

To call a function block, it is commonly required to set the `AXISGROUPIDX`, which defines the robot to communicate with and to call the function block's `OnCycle()` method. The communication process is started by a datagram that is cyclically sent by the robot. The `KRC_READAXISGROUP` class needs to be instantiated to process the data coming in. Once data is received by the UDP socket as a `byte[]` array, that array is set as the class' `KRC4_INPUT` and `OnCycle()` is called. The robot now needs to be initialized through the `KRC_Initialize` function

block. KRC_WriteAxisGroup formats the data that can be sent back to the robot via the UDP socket communication.

KRC_WriteAxisGroup also raises an event that informs all subscribed event listeners about the robot's current axis and Cartesian position, the state of its inputs and outputs, its buffer size, and various debugging information, allowing the host software to visualize the robot's current position.

Once communication has been established, it is possible to access cyclic data such as the current axis position via KRC_READACTUALPOSITION. Once the bits of the robot's Automatic External interface are correctly set via the KRC_AUTOMATICEXTERNAL function block, it is possible to interact with the robot.

For example, KRC_READANALOGINPUT gets the value of an analog input. Shortly after the component's EXECUTECMD field is set to true, and OnCycle() is called, the class' Done field changes to TRUE, and KUKA|prc can access the analog value as a float from the Value field. To query the analog input again, the class must be instantiated again, repeating the process. As such, it is important to look at the results and the state of the classes to ensure that components are correctly instantiated and the calculation has finished.

For dealing with larger toolpaths, every interaction mode (see below) sets a specific buffer size. The buffer is implemented as a list of PRC_mxA_Command, whose OnCycle() method gets called at every cycle. If the buffer is empty, a new command is added, with EXECUTECMD set to true. Once its Busy field is true, the following command in the toolpath is added until the list has reached the desired length. Once the Done field of a command field is true, EXECUTECMD is switched to false. At the next cycle, commands without an active execution parameter may be safely removed from the list and are no longer called cyclically.

When the data is refreshed, while there are still commands in the buffer, the BUFFERMODE of the first new command is set to ABORTING, automatically canceling all other previously active commands and making it save to remove them from the list in KUKA|prc.

In parallel, the state of all commands is checked to detect inconsistencies. In early versions of mxAutomation, it was sometimes not correctly signaled that commands had finished. Therefore, a plausibility-checker searches for the current commands whose `ACTIVE` output is true and flags all previous commands for removal.

When the buffer runs empty, the robot controller can no longer blend movements, as the subsequent movements are unknown, leading to jerky movement with constant stops. This happens when the density of a toolpath is so high that the robot reaches positions faster than the 256-byte cyclical communication can transfer them. KUKA|prc, therefore, offers the option of having the robot's 0-100% override speed dynamically reduced when the buffer is at the risk of running low, thus increasing the cycle time but ensuring a smooth toolpath.

The concept of mxAutomation was to develop a "driver" to control KUKA robots in real time. As demonstrated above, the library integrates only the most basic functionality and leaves most details to the user. This is probably because their core product is aimed at PLCs, making the C# and C++ library a byproduct for customers with special requirements. While the direct translation of structured text to C# makes for a relatively unintuitive syntax, the much more modern Java-based Sunrise platform offers a similar way of programming blended robot movements by pushing movement commands into a background queue (Chapter 4.7).

4.6.4 Strategies for Realtime Interaction in Visual Programming

Understanding the data flow and internal operation is essential for recognizing the potential and limitations of mxAutomation. However, within the scope of visual programming, there are significant architectural challenges towards implementing real-time control due to the directed nature of the graphs, where data is only flowing from left to right, and the complex interplay between cyclical data (e.g., constant refreshes by a 3D camera) and the mxAutomation client.

If mxAutomation's real-time control is used to simply "stream" toolpaths to the robot, rather than, e.g., copying the data in the form of an SRC file off a USB stick, the return data only provides a real-time visualization without any data-flow-related implications. However, problems arise

when the robot's data is supposed to inform the parametric design process, e.g., to correctly set the position of a robot-mounted 3D camera in the world coordinate system. Such a data flow can no longer be natively represented in Grasshopper. However, "underneath" the visual programming surface, there are, of course, multiple ways to pass data.

The chosen solution is intended to provide a native user experience while circumventing Grasshopper's restrictions against cyclical data flows: The main mxAutomation component collects the data (commands, robot, tool) and UI settings and uses them as the inputs for the `KUKAprc_mxA` class' main method. Its only data output contains a reference to that instance. The data display component takes that referenced object and subscribes to an event called whenever new data arrives from the robot. Even if the connection between the main and display components is disconnected, the display component is still subscribed to the event and will receive notifications.

A significantly more challenging aspect of real-time interaction within a visual programming environment is the handling of cyclically refreshed data in an intuitive and efficient way. As with the offline simulation, KUKA|prc compares the hash value of all input commands to the buffered hash value (see Chapter 4.3). Therefore, no action is taken if the collected input data is identical to the current data set. However, if the data differs, the reaction of KUKA|prc's mxAutomation client depends on the buffer mode (Figure 29), of which there are currently five options:

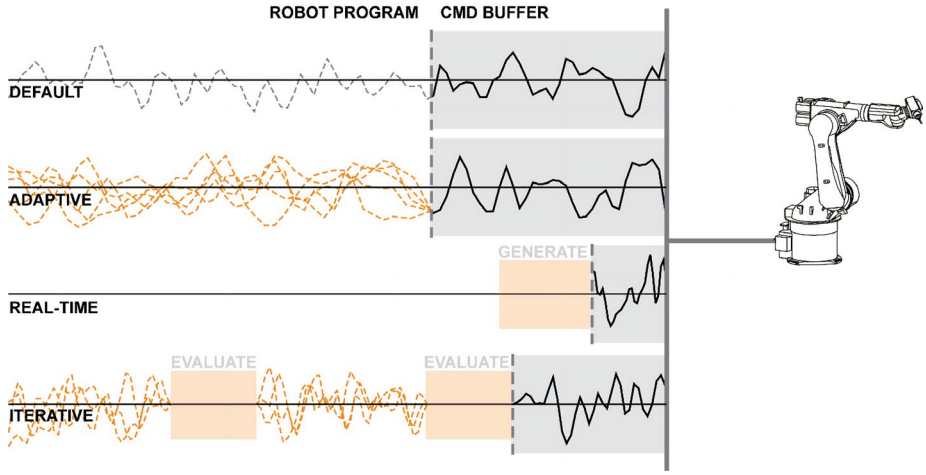

Figure 29: Four of the mxAutomation control modes developed for KUKA|prc

Default Mode uses a relatively large buffer of 30 commands. A new set of commands immediately overrides the previous data set, i.e., the first new command uses the Aborting mode, canceling previous commands and causing the robot to stop and directly move to the first programmed new position. This mode is primarily used to stream a list of commands to the robot without any specific interaction.

Iterative Mode is a variation of the Default Mode that always finishes a data set before starting a new one. Therefore, at the end of a data set, the client will either retrieve the most current new data set – discarding any older ones – or stop and wait until new data is provided. An example of the Iterative Mode would be integrating a cyclical data source, such as a camera performing object recognition for pick and place applications, where the robot would attempt to manipulate the most recently detected object.

Queue Mode is similar to Iterative Mode in that it always finishes a data set. However, unlike Iterative Mode, all data is reliably queued and never discarded. As long as there are data sets, the robot will process them. An example of Queue mode would be an ordering process at a store, where customers submit orders that need to be fulfilled by the robot.

Adaptive Mode uses a significantly smaller buffer to ensure a short reaction time. Whenever a new data set with the same number of commands is received, the client will keep all commands that have already been passed into the buffer unchanged but stream any new commands from the latest data set. If the size of the data sets differs, the client can optionally also continue with the new command closest to the most recent buffered position. This allows responsive processes that react to changing input parameters. For example, a milling process encountering high vibration could not just change the process speed but parametrically reduce the maximum stepdown accordingly, allowing complex, parametric processes.

Real-time Mode uses the smallest buffer. The data set may only consist of a single-axis position. At every cycle, a new target position between the most recent buffered position and the current target position is calculated. The delta in axis values between each position can be defined in the settings. The robot does not need to wait until it has arrived at the programmed position but can react immediately to new input. Due to its

implementation, Real-time Mode is not meant for accurate machining tasks but for more playful interaction between man and machine.

In a traditional, text-based programming environment, these interaction modes would not add a significant value to the user, as they could mainly be represented by a while loop that checks the buffer size and adds additional commands whenever a threshold is passed. However, within a visual programming environment that neither supports loops nor provides an accessible way to utilize conditional clauses, a user can rapidly create interactive programs that enable individualization and mass customization.

A particular problem is that of data persistence between iterations. In the case of a brick-stacking application, it is desirable to record the positions of the deposited bricks. While Grasshopper provides a data recorder, it does not filter out duplicates. If cyclical data coming from either the robot or an external sensor refreshes the graph, even if the data itself does not change, a new data point is recorded, possibly leading to thousands of data points over a minute and ultimately decreasing the performance of the visual programming environment.

Therefore, any such task requires an application-specific code to track its progress. An example is presented in Chapter 5.4.

4.7 New Robot Architectures

Alongside KUKA robots using KRC1-KRC5 controllers, KUKA|prc also supports the more recent KUKA Sunrise environment. While KRC controllers use KUKA's proprietary KUKA Robot Language, Sunrise builds upon Java, enabling programmers to use state-of-the-art programming patterns and implement external libraries. Sunrise has been implemented through the DIANA project (Figure 30) (Braumann, Stumm, and Brell-Cokcan 2016) and is also used in Chapter 5.2.

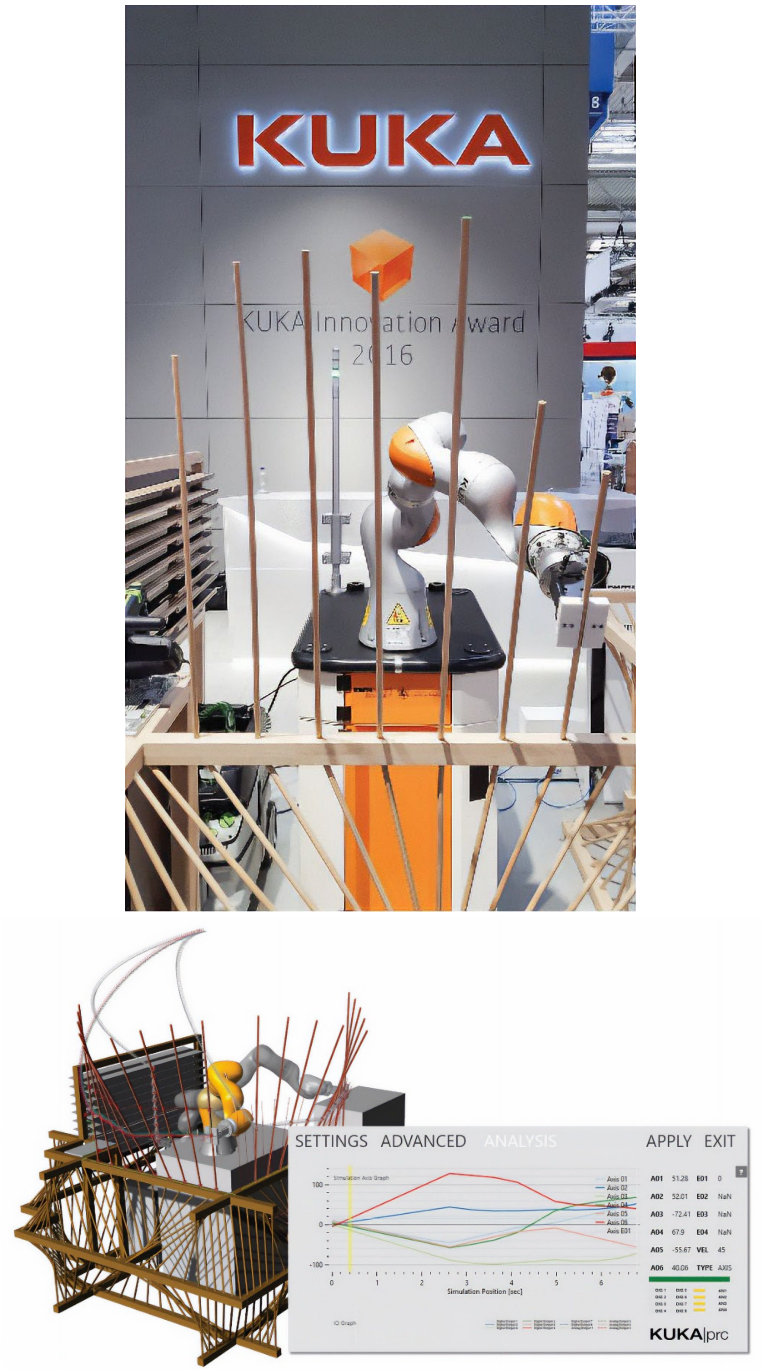

Figure 30: DIANA project using a collaborative KUKA LBR iiwa robot with KUKA|prc at Hannover Fair.

4.7.1 Data Preparation for Sunrise

When using Sunrise, projects are no longer copied to the controller as files but uploaded via a Git-like process through the Sunrise Workbench, an integrated development environment (IDE) based on the open-source software Eclipse.

As such, generating files containing Java code directly for each project is not feasible. Instead, KUKA|prc serializes the list of commands in an XML structure (Figure 31). A custom KUKA|prc Java library supporting the deserialization of XML data into the relevant Java classes is provided for Sunrise.

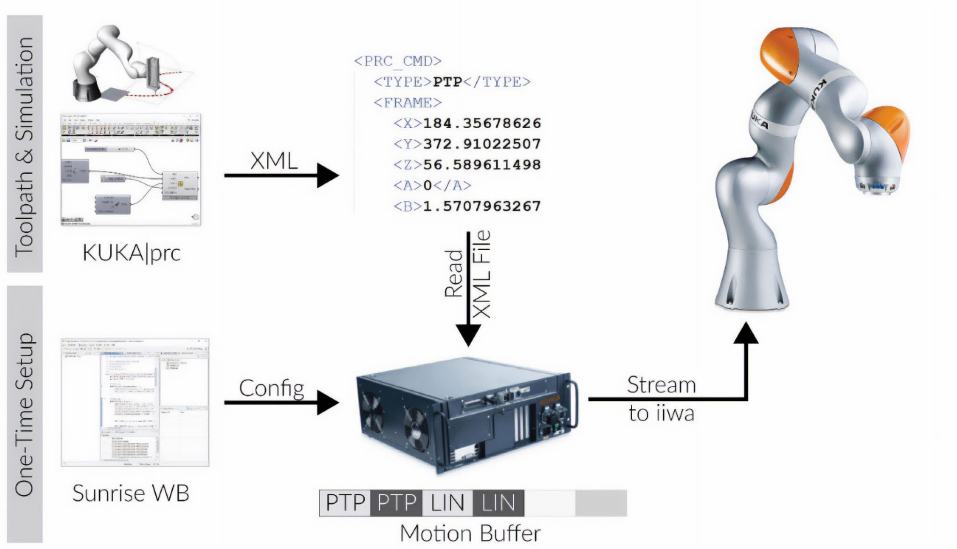

Figure 31: Controlling a KUKA LBR iiwa robot running Sunrise.OS through KUKA|prc.

Currently, Sunrise only supports the LBR iiwa series of collaborative robots, which add an additional axis between axis 2 and axis 3 for a total of 7 axes. KUKA|prc addresses the axis not as axis 3 (which would make the flange axis 7), but instead refers to it as E1, similar to how KUKA referred to the redundant axis of the iiwa's predecessor LWR-4.

The redundant axis, along with more minor changes like the different starting point of A2 – which makes the robot perfectly vertical when all axes are set to 0 – require an adapted kinematic solver for the iiwa's forward and inverse kinematics and the rendering process.

The solver was further expanded to support the KUKA KMP 200 mobile platform. By setting E2, E3, and E4, the user can define the mobile platform's absolute X and Y position and its rotation around its axis, referred to as Theta (Figure 32). This functionality goes beyond the regular kinematics solver, where a robot may only be translated but not rotated by an external axis. For the code output, the absolute values are segmented into a series of relative movement commands, as absolute coordinates require a known model of the KMP's environment through its Sick laser scanners. To switch between the regular and seven-axis kinematic solver, the `PRC_RobotData` class' `PRC_KinematicType` parameter has to be set to either `6DOF` or `7DOF`.

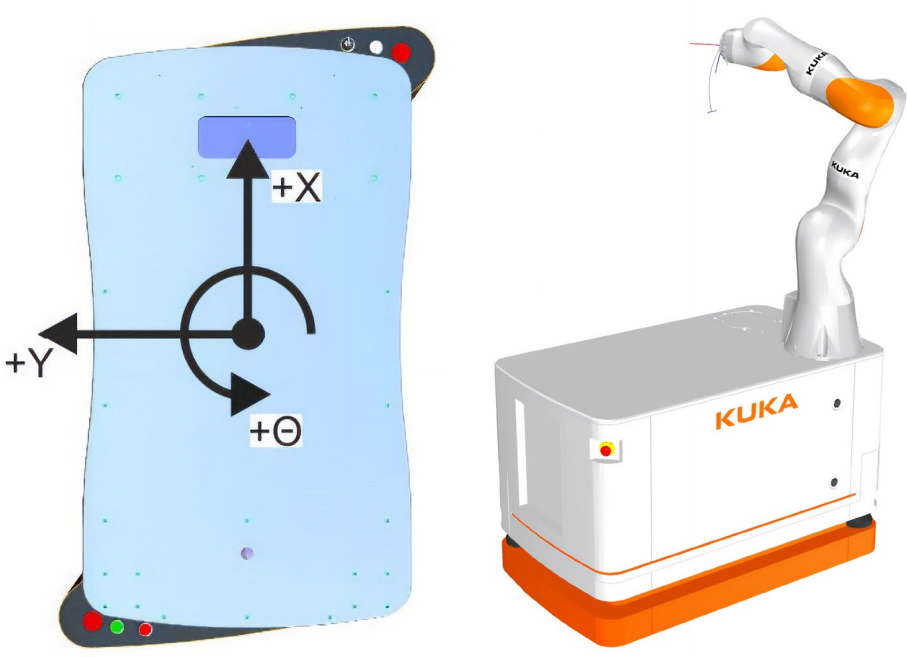

Figure 32: KMP 200 platform coordinate system (left, KUKA Robotics) and KUKA|prc implementation (right).

4.7.2 Sunrise Integration

For the Sunrise controller, KUKA|prc provides a template for a Sunrise application, deriving from the `RoboticsAPIApplication` class, where all the base information is set, like the reference to the robot (`LBR`) and controller (`Controller`), as well as `strings` defining the name of the tool and its tooltip, as well as the base coordinate system. Once the

tool and base have been set at the robot, they can be extracted from the application data via the `getApplicationData()` method and the according name, e.g., `getApplicationData().getFrame("/BASE1")`.

The library queries the file location of the XML file, either via a `JFileChooser` dialog or as a static `string`, and then uses the `core_ReadXML` method to deserialize its contents, creating native Java classes for each KUKA|prc command. The `prc_Core.core_Run` method takes the robot, tool, base, and command data and streams it to the robot.

For non-blended movement, where the robot would stop briefly at any given position, the process of getting the robot to move is relatively straightforward, e.g., for a Cartesian movement, the tool center point (actTCP in Code 9) is instructed to proceed to a given position with a given speed an acceleration (Code 9, line 1). The syntax is similar for axis-defined movements, with the difference that they refer to the robot and not the TCP (Code 9, line 2).

```
1
actTCP.move(lin(cmd.linMove.frame).setCartVelocity(cm
d.linMove.vel) .setCartAcceleration(linacc));
2 robot.move(ptp(cmd.axisMove.axispos)
.setJointVelocityRel(cmd.axisMove.vel).setJointAccele
rationRel(ptpacc));
3 IMotionContainer mc = robot.moveAsync(…);
4 motionContainers.add(mc);
```

Code 9: Movement programming in Sunrise.

Additional motion parameters like `setMode` allow further movement customization, e.g., setting up a `CartesianImpedance` control mode that parametrizes the robot's stiffness in various degrees of freedom.

For blended movements, similar to what the `C_DIS` or `C_PTP` parameter would set for KRC 4, the complexity increases as the robot needs to know at least one or more subsequent to blend between them smoothly. In Sunrise, that is implemented via the `moveAsync` method, which uses the same syntax as the regular `move` but adds blending parameters such as

`setBlendingCart`. The core difference between them is that `moveAsync` is not blocking, so the code does not wait for the movement to be executed.

However, it is essential to know how many commands are currently buffered. As such, the library keeps a reference to the asynchronous motion in a list of `IMotionContainers` (Code 9, lines 3-4).

By waiting until the last `IMotionContainer's isFinished()` method returns true, it is possible to ensure that subsequent commands, e.g., cause the robot to wait or trigger an output, are activated at the right time on the toolpath.

Finally, KUKA recommends that highly complex toolpaths be collected into a so-called `MotionBatch`, a list of movement commands that get executed simultaneously. Therefore, any sequence of movement commands –uninterrupted by, e.g., digital outputs or wait commands – is added to a `List<RobotMotion<?>>` and then used as a constructor for the `MotionBatch`.

4.7.3 Realtime Control

An advantage of Sunrise over the older KRC architecture is the platform's openness. While the KRC platform can send and receive arbitrary Ethernet data via TCP or UDP using the KUKA Ethernet.KRL tech package parsing and working with this data in the older KRL language is complicated. Interfaces that do not just provide generic data transfer but also application-specific functionality like mxAutomation are valuable additions to the KRC architecture but, in turn, are not easily expandable with new functionality.

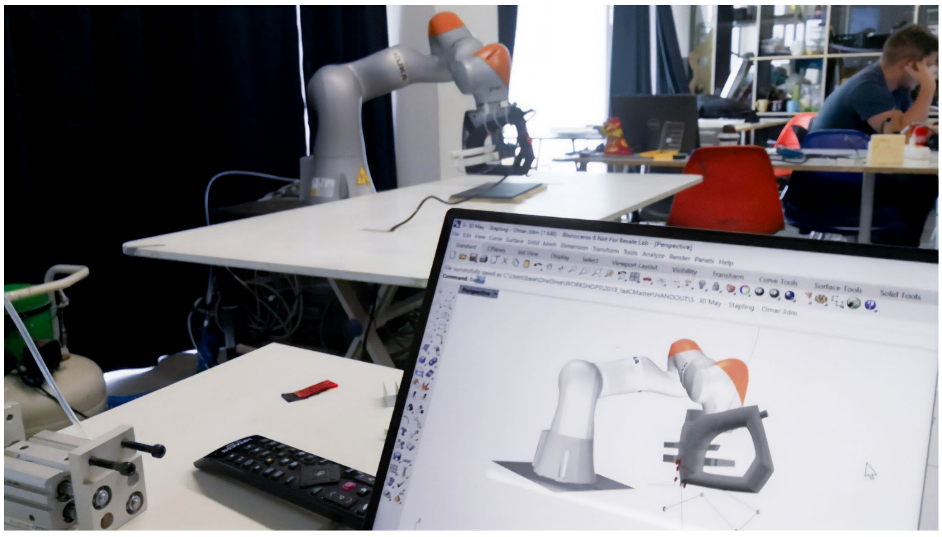

Figure 33: Realtime control of a KUKA LBR iiwa at a workshop at MRAC, IaaC, Barcelona.

On the Sunrise platform, KUKA|prc can provide a mxAutomation-like user experience in a much simpler way through built-in Java and C# socket communication. Within KUKA|prc, the `PRC_SunriseCommunicator` class is built around a `UdpClient` communicating via Ethernet with a connected Sunrise controller.

Whenever the `PRC_SunriseCommunicator` receives new data, it adds it to a `ConcurrentQueue`. The `UdpClient` is running as a separate task that dequeues the data, formats it, and sends it as a UDP datagram to the Sunrise controller. When it receives data, the socket's `AsyncCallback` calls a function that enqueues the received data into another `ConcurrentQueue` to be displayed.

On the Sunrise side, the `core_UDP` method creates two threads for sending and receiving UDP data, each assigned to a different socket. When the input thread receives data, it parses it and creates the corresponding `PRC_CommandData` Java objects, which are then enqueued in a `LinkedBlockingQueue`.

The main Sunrise program waits until elements are enqueued, dequeues them, and has the robot execute them immediately. The UDP sender-thread is provided with references to the queue as well as the LBR object so that it can cyclically send the current axis/Cartesian position and the

number of buffered commands back to KUKA|prc, where it can be visualized with the regular forward kinematics solver.

KUKA|prc also provides an optional variation of the control interface, which works identically on the KUKA|prc side while using the same threads for communication on Sunrise. However, rather than using standard movement commands, it builds upon KUKA Sunrise.Servoing, a tech package that was developed to correct robot toolpaths in "soft" real-time, i.e., non-deterministic applications (KUKA 2015).

However, this correction can also be applied continuously to create responsive robotic applications. While the previously presented solution processes all positions in the buffer before moving to the most recent position, Servoing allows the robot to change its toolpath immediately according to the new data.

Concerning the data flow, this means that an entire toolpath can be sent via Ethernet to the robot through regular UDP communication. At the same time, for Servoing, it only makes sense to continuously send the target position, as all but the most recently received position will be automatically discarded. An exemplary application is presented in Chapter 5.4.

5 Skill Digitization

As presented in Chapters 3 and 4, KUKA|prc has been developed as an accessible robot programming environment whose architecture is directly optimized to enable non-standard robotic applications with a focus on mass customization. A specific interest lies in transferring existing skills from a person to the robot, a process that industry generally refers to simply as automation, when a machine replaces a worker to perform a simple task.

The author introduces the term "Skill Digitization" (Braumann 2020) as a participative process that does not replace human labor but adds value to a process by combining robotic strengths with process knowledge. The approach towards skill digitization is twofold: Through knowledge transfer and education initiatives, process experts should be guided towards being able to define, program, and run robotic processes by themselves. However, very often, the complexity of the processes is too high to expect a new user to be able to realize complicated robotic tasks. In that case, skill digitization aims to develop the process with a highly participative approach and use bespoke, user-oriented interfaces that allow process experts to implement their knowledge and thus adapt the robotic task.

The following applications of skill digitization cover a wide field of applications in the creative industry and beyond (Braumann and Brell-Cokcan 2018), from directly duplicating a dancer's captured movements with a complex multi-robot setup to innovating craft processes and implementing new sensors and data sources.

5.1 Process Duplication

The most direct approach to automation is the duplication of existing processes. In industry, the universal motivation behind automating an existing, manual process is improved cycle times and higher accuracy. In contrast, the following two projects do not improve existing processes but replicate them to the millisecond and millimeter, bringing human movement patterns directly to a machine.

Doing so does not make them very efficient machines for fabrication but generates performances with a particular aesthetic that can be played on demand. However, neither robot programming software nor the robots are optimized for such purposes, requiring the development of new control strategies.

5.1.1 Robotic Marionette

The marionette project was developed for the new permanent exhibition at the Ars Electronica Center in Linz, which refers to itself as the "Museum of the Future." Its concept was to allow two KUKA robots to play a marionette. Several papers from the mid-2000s present approaches to control marionettes (I.-M. Chen et al. 2004; Martin et al. 2011). However, these projects turn the marionette itself into a string-controlled robot, while the Ars Electronica's concept wanted us to control marionettes the same way as a human controls them.

Thus, a marionette control bar is mounted directly onto each robotic arm. While this allows the robot to manipulate the control bar with six degrees of freedom, marionette players commonly also directly interact with the strings using their second hand, an option not available to a robotic arm unless additional motors are used.

The initial consideration to map human kinematics to a marionette's kinematics through motion capture was discarded, as the degrees of freedom are insufficient to express the recorded human motions directly. In contrast, a human puppeteer can intuitively exaggerate movements to make emotions more visible.

To achieve that goal, it is necessary to solve the following challenges:

- Import the motion-capture data into Grasshopper.
- Postprocess the data to fill gaps and fit it into the robots' workspace.
- Devise a way for the robots to duplicate not only the path but also the timing of the captured data.

For motion capture, Ars Electronica's OptiTrack motion capture system is used, capturing three reflective spheres that were mounted at the position of the robot's flange (Figure 34, left). Calculating the control bar's position and orientation in 3D space is possible through these three points. This data is then exported as a CSV file containing the position of the rigid body – i.e., the compound object defined by the three points – and the individual marker at 100 frames per second, i.e., a 0.01-second timestep.

However, the rigid body may lose track when one marker is obstructed. This happens especially when the number of motion capture cameras is too low for the given task.

Figure 34: Motion capture for the marionette project (left), initial prototypes on the robot (middle), and final project (right).

Rather than using the compound object, the individual marker data is therefore used to generate the coordinate system objects, representing the movement of the control bar. However, the OptiTrack software creates a new data field whenever tracking is briefly lost because the reflective spheres cannot identify themselves. For a two-minute take, 12 markers were recorded instead of 3 markers, probably due to losing track 9 times within the timeframe.

It is, therefore, necessary to parse the raw CSV data and reconstruct the set of three points. Usually, this can be achieved by simply matching a point to the closest point of the previous timestep; however, if data is missing, the empty space needs to be interpolated. This is done linearly

by searching for the next complete set and constructing a line from the previous known to the next known position.

A plane is then mapped through the set of three points, allowing a refinement of the interpolation by looking for the intersection between the known distance from one known position and the similarly known distance from the other known position. A similar algorithm is used when tracking all markers is lost, recreating the order of the set of points based on the known angle and distance between them. Once the trajectories are computed, they can be placed within the robot's workspace.

However, it is not sufficient for the robot to trace the captured toolpath: Maintaining the original timing is crucial to duplicate a marionette performance. While industrial robots allow the user to set the target, movement type, speed, and acceleration, it is by default not possible to tell the robot when it should arrive at a target position, as robots generally try to perform a task as quickly as possible.

If a trajectory contains multiple points with the same position, indicating that the puppeteer did not move, the robot will skip them and continue with the toolpath rather than wait. For animation, KUKA provides the ready2_animate software package, a high-end tech package aimed at the entertainment industry.

This functionality can also be replicated using available technology. At the core of time-guided robot programming is KUKA's Realtime Sensor Interface (RSI, see Chapter 4.6), which controls a robot cyclically in real time. This process cannot be reliably realized in a non-real-time operating system like Windows. As such, the two KUKA robots are connected to a Beckhoff PLC running TwinCAT 3 using two master-master couplers, allowing a cyclical exchange of process data in hard real-time.

Consequently, the entire path-planning and interpolation process must occur within the visual programming environment towards preparing a final CSV file at 1000 lines per second. The toolpath is placed within the robot's workspace and transformed to optimize reachability and avoid singularities. The data is then linearly interpolated by a factor of 10 to reach 1000 lines per second, and the resulting plane objects are passed

to a *PTP* component and then to the KUKA|prc *Core* component, which performs the inverse kinematics calculations. Rather than using the SRC file, the Analysis output providing the axis position for each plane is used to deliver the data formatted as comma-separated values and saved to an external file.

This CSV file containing 12 columns of data – representing two robots with six axes each - with one row per millisecond of performance, is read by the real-time PLC system and streamed via EtherCAT at a speed of one row per millisecond to the connected robots.

Ensuring no abrupt movements can damage the robot within this workflow is essential. Sudden movements can be caused by the actual movement of the puppeteer or by interpolation problems during tracking. Data may be smoothed at three stages within the workflow: Cartesian data at the original 100 frames per second, Cartesian data after the interpolation to 1000 frames per second, or the axis values after the inverse kinematics solver.

The best result was achieved by interpolating directly at the axis level, as the separate interpolation of the plane's origin-point and X / Y vectors led to inconsistent plane orientations. For the interpolation, a component implementing a Butterworth filter was developed and integrated, as Grasshopper does not provide any signal processing filters (Figure 35).

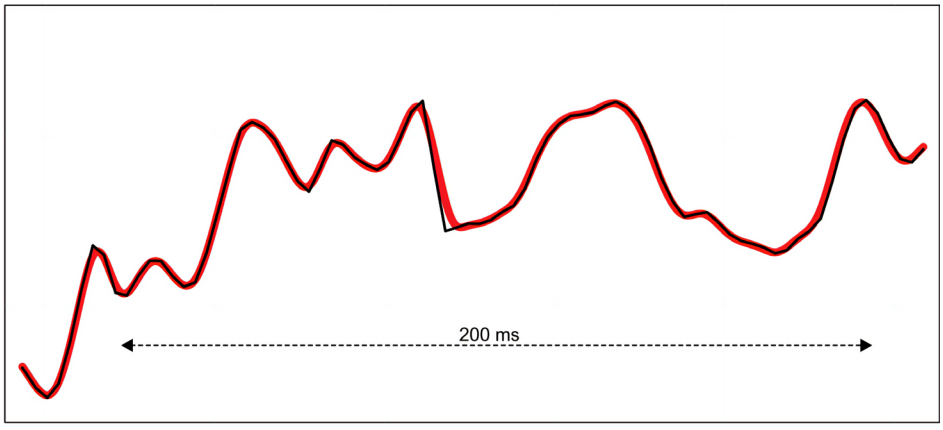

Figure 35: 0.2-second choreography excerpt plotting A1 movement generated through the inverse kinematics of the motion capture data (black) and the smoothed and interpolated toolpath sent to the robot (red).

Knowing the axis position and the timestamp of each position now enables calculating the speed and acceleration at any given millisecond position. While KUKA provides the maximum speed of all robots within the data sheets, the acceleration had to be derived from KUKA's internal motor parameter files.

For example, according to the machine data, A1 has a maximum speed of 300 degrees per second and reaches that speed within 468 milliseconds, thus achieving an acceleration of 641 degrees / sec ^{2}. This makes it possible to check each position to ensure that the robot can perform a given trajectory.

Finally, the data has to be reformatted, as RSI only uses relative data, with the initial robot position as the starting point. Therefore, the robot's start position is subtracted from the axis values. Furthermore, by default, the PLC would only transfer unsigned integers. As such, 360 is added to the axis values to ensure that only positive numbers are transmitted. The values are multiplied by 10000 and then cast as integers, thus keeping a sufficient 4-digit accuracy. These transformations must be reversed within RSI to get the actual axis values.

The resulting performance closely matches the motion-captured data. It greatly benefits from the noticeable "human" influence derived from the motion-capture process, which differs significantly from a purely animated process.

Visual programming and KUKA|prc was crucial towards realizing the marionette project, as it was possible to very rapidly integrate new algorithms for importing, postprocessing, and analyzing data and to evaluate the results more or less in real-time, thus being able to make informed decisions regarding the viability of a trajectory quickly.

Credits: Concept and programming were done by the author, motion capture was supported through the Ars Electronica and Amir Bastan, and Katharina Halus contributed as puppeteer.

5.1.2 Underbody

The Underbody Project is a collaborative effort with Silke Grabinger to capture Silke's movement and bring it onto the stage through five connected robots, one of which is a heavy-duty Fortec-series robot that performs global movements. In contrast, the four attached Agilus-series compact robots are responsible for one joint. Geometrically, the shoulders and hips form a coordinate system that sets the global position of the large robot. The small robots only need to cover the movement of the joints in relation to the shoulder-hip coordinate system.

The robotic "puppet" is intentionally represented in a minimalistic way through several welded, powder-coated steel rods to not distract from the robots, whose movement is an integral part of the performance (Figure 36).

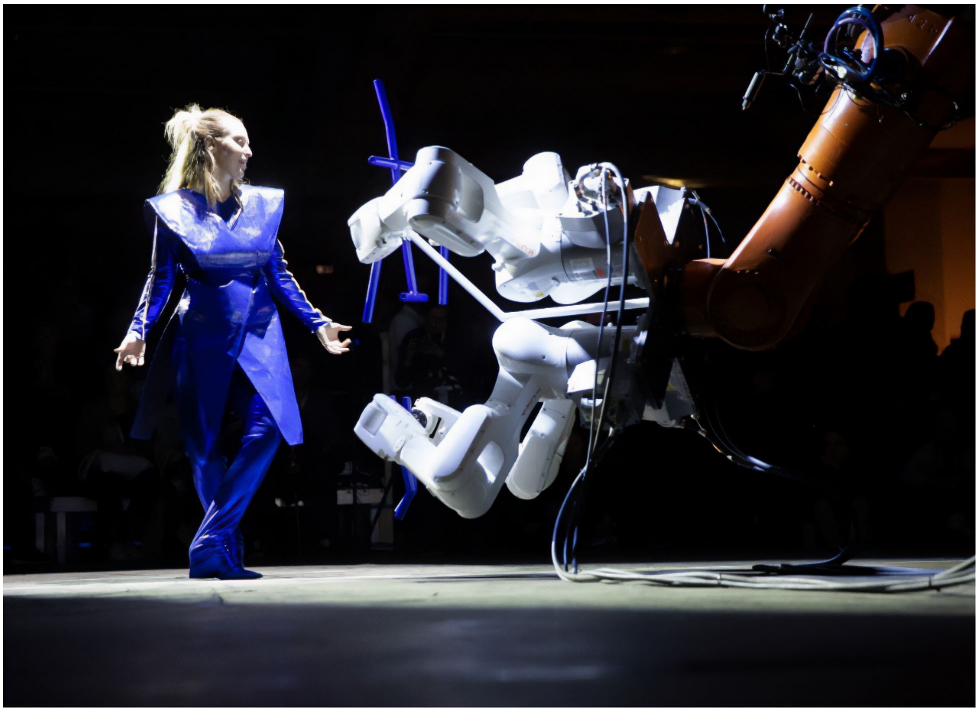

Figure 36: Robot setup of the underbody project, using five synchronized robotic arms.

Where the Robotic Marionette project (Chapter 5.1.1) deals with a very limited movement range, the high mobility of the Underbody project's robots leads to several new challenges:

- Feedback must be provided to the dancer in real time during the motion capture process
- Several degrees of freedom of the robotic system must be identified and used to improve reachability and remove the chance of collisions.

Controlling the robots is achieved identically to the previous project by streaming data from a CSV via a high-end Beckhoff PLC running at a one millisecond cycle time to the robots. However, while the marionette project required just 12 rows of data, the Underbody Project's five robots require 30 rows with a significantly longer performance duration. The final data flow is outlined in Figure 37.

Figure 37: Underbody data flow.

Early simulations of the dance performance in the KUKA|prc environment showed that reachability would be a significant concern of the project, as the motion range of an adult exceeds the motion range of the available KUKA Agilus-2 robots with their 900mm reach. For the motion capture process, the dancer is therefore not just captured by the OptiTrack system but also by a Microsoft Kinect 3D camera whose data is processed in real-time and displayed in Grasshopper. This allows the dancer to check her movement in real-time to ensure that the small robots can reach every joint. Therefore, the dancer can get an intuitive understanding of the constraints of the robotic system so that it becomes possible to stay within its limits most of the time.

Figure 38: Live robot pre-visualization during motion capture through KUKA|prc and a Kinect-2 sensor.

Ultimately, the Kinect's data was used for the final performance, as the OptiTrack system could not provide reliable tracking of the dancer due to an imperfect setup and lower-quality cameras compared to the Robotic Marionette project.

The postprocessing of the data is done through a range of custom components implemented in Grasshopper. Initially, the data coming from the Kinect is provided as a CSV file with 101 columns of data, representing 25 points of the skeleton tracking, each represented by an X, Y, Z, and a confidence value, as well as a timestamp in ticks, where 10000 ticks represent one millisecond. The algorithm removes any points with low confidence, replacing them with interpolated data, and performs Butterworth filtering to remove jitters from the captured data, e.g., due to the low depth resolution of the Kinect.

To be able to optimize a process, it is necessary to identify degrees of freedom where the optimization may take place. Based on the skeleton data, a parametric model is built in Grasshopper that exposes several degrees of freedom (Figure 39):

- The general position and orientation of the skeleton in relation to the shared base coordinate system of the small robots.
- The position and rotation along the joints where the robot would grasp the joint.
- The joint's offset from the robot's flange.

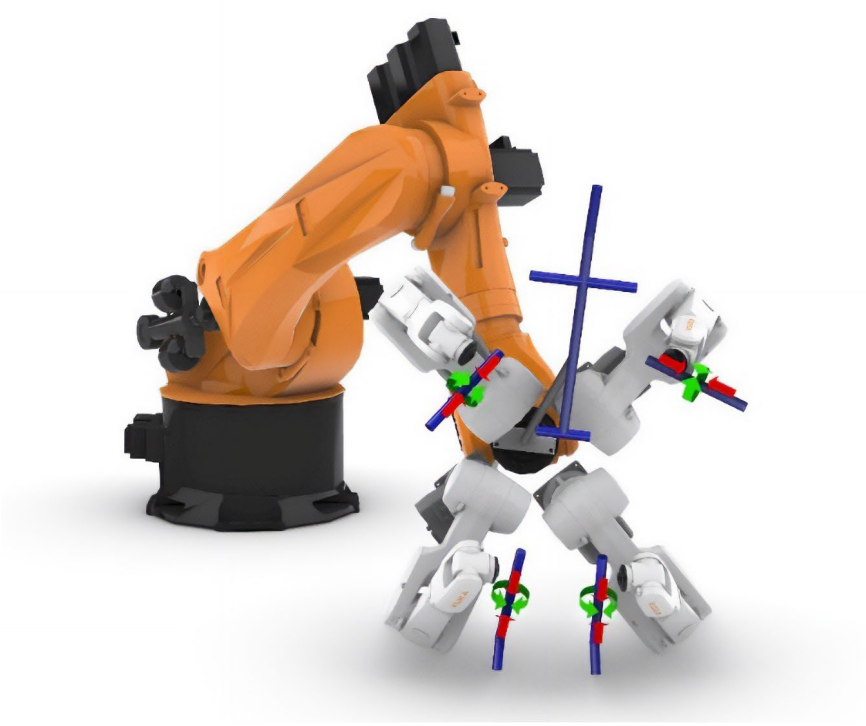

Figure 39: Degrees of freedom of the developed robotic system for optimization.

An initial approximation is found through an intuitive interaction of the user with the different parameters; however, for the fine-tuning, Grasshopper's integrated evolutionary solver Galapagos is applied, defining a fitness function that would keep the robots' axes within their range and minimize collisions between the robots and the joints as per the Analysis output of KUKA|prc.

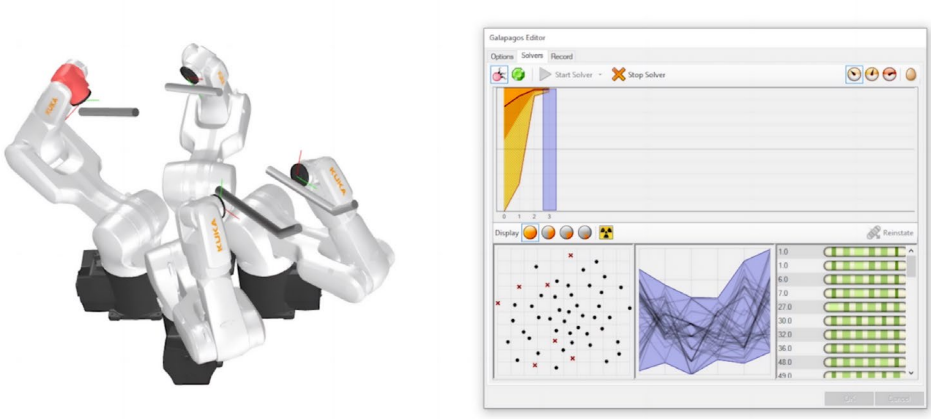

Figure 40: Reachability optimization through the evolutionary solver Galapagos.

As even the evolutionary solver was not able to find a perfect solution within several hours of computational time, the component performing the millisecond interpolation and smoothening is expanded with two additional functions:

One would limit the maximum translation per timestep to keep velocity and acceleration within specifications, while the other would cap the axis values to the default range. These methods cause a slight deviation from the motion-captured data; however, as they are only used in a few extreme cases along the 5-minute-long choreography, the approach offers an acceptable tradeoff between accuracy and safety.

The five-robot setup used for the Underbody project represents a highly complex system that would be extremely hard to understand and fine-tune without the graphical, interactive approach of visual programming. Other commercial simulation software also does not allow a simulation of robot-mounted robots. Therefore, KUKA|prc uniquely enables an immediate visualization of geometric data and graphs representing the values of the different axes for analysis and optimization.

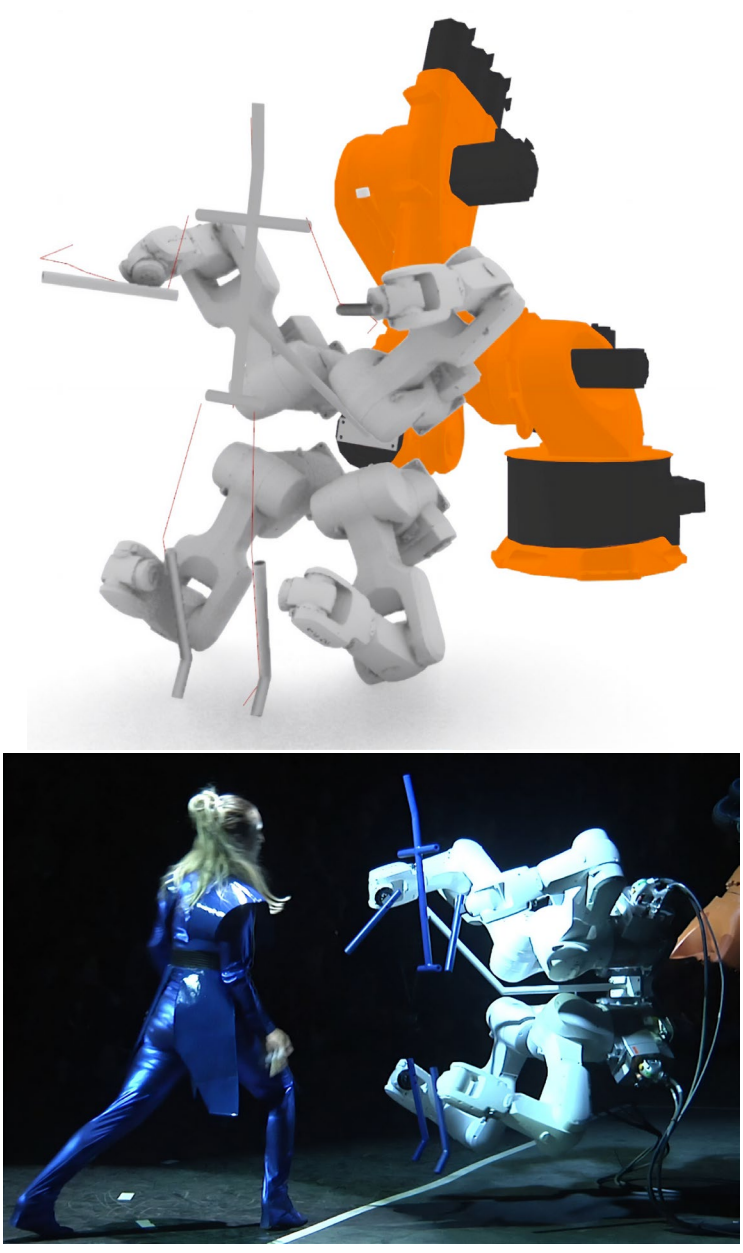

Figure 41: Process visualization (top) and final project (bottom).

Credits: Concept and programming were done by the author, motion capture supported by Peter Freudling and Roland Aigner, and performance by Silke Grabinger.

5.2 Process Augmentation through Analysis

A core aspect of digitizing an existing process is a deep understanding of its requirements and limits. The following projects represent complex craft processes that go beyond the scope of CAD-CAM workflows and combine the craftsperson's material expertise and process knowledge with the speed, accuracy, and flexibility of robotic arms. Visual programming combined with KUKA|prc's robot simulation and control capabilities is demonstrated to be a highly accessible tool that allows craftspersons to either quickly change basic process parameters or delve deep into code to optimize robotic processes.

5.2.1 Robotic Stone Chiselling

The AROSU project (Brell-Cokcan and Braumann 2015; Steinhagen et al. 2016) was funded through the European Union's FP7 framework to develop a robotic process for stone structuring on behalf of Bamberger Natursteinwerke, a SME based in southern Germany (Figure 42). The consortium involved many partners, with the research primarily headed by TU Dortmund (mechanical engineering), Labor (electrical engineering), and the Association for Robots in Architecture (software development), where the author was acting as a primary researcher.

While chiseling with a robot is mechanically challenging due to the forces acting on the robot's gears, a primary challenge of the early research phase was the lack of a proper process description, as neither scientific publications nor practical manuals elaborated on details such as the process forces. Existing pneumatically actuated chiseling devices did not generate results close enough to the manual process.

That made it necessary to solve the following challenges:

- Get a clearer understanding of the craft process.
- Solve mechanical challenges towards chiseling with a robot.
- Develop software for robot chiseling.
- Investigate ways of providing added value through the robotic process compared to manual labor.

Figure 42: Stone chiseling process developed through the AROSU project.

The need for a machinic solution comes from the lack of skilled labor, as even the regular chiseling of a stone surface requires expert knowledge to achieve a good surface finish; however, at the same time, the mason's expertise may also be used for higher-value and less repetitive projects.

Bamberger Natursteinwerke was already an experienced user of industrial robots, having used four on-site programmed KUKA robots for milling natural stone as well as palletizing. Therefore, building upon that experience towards using robots for chiseling was a logical step.

The first step of the project involved an in-depth analysis of the chiseling process in close collaboration with the process experts, measuring forces with a force-torque sensor and tracking the movement of the chisel in 3D via two high-speed cameras (Figure 43) (Steinhagen, Brüninghaus, and Kuhlenkötter 2015). The resulting data showed that a manually actuated chisel does not simply move into the stone in a straight line but rather "excavates" material as the hand cannot perfectly guide the tool, creating the desired look of the strokes compared to existing machinic applications. This captured data then guided the development of a novel actuation tool by TU Dortmund and informed the software interface developed by the author.

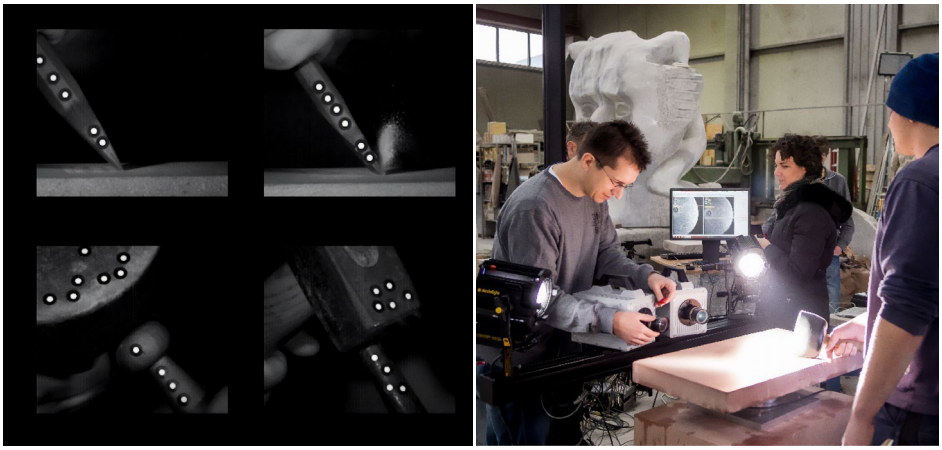

Figure 43: Analysis of the stone-chiseling process through the AROSU project.

On the software side, the negative volume of the stone subtracted through the chiseling process is defined via an algorithm, taking the geometry of the tool, the process force, and depth into account, which was then implemented into the visual programming environment. The user can now switch between a simplified representation that displays every stroke as a line, whose length is derived from the chiseling tool, or an actual 3D representation. The code also allows us to export the surface finish as a greyscale depth map for easy visualization.

While the immediate goal of the research project was to realize planar, regular stone chiseling towards facilitating the masons' work, several use cases were identified where the robot provides an added value beyond cycle time. In particular, an efficient programming strategy would enable the robot to process individual patterns with the same efficiency as the regular pattern.

Therefore, a large part of the research evolved around the definition of new patterns, e.g., by rotating the edge of the chiseling tool depending on the brightness of a bitmap or the proximity to an orientation point. A significant advantage of visual programming within that context is its immediate feedback concerning aesthetic output and robotic reachability, enabling intuitive interaction with the parameters.

Due to a mistake in the parametrization of a coordinate system during testing, a rotation axis was accidentally wrongly connected, rotating the chiseling tool's coordinate system out of the plane representing the

stone surface, essentially creating a five-axis chiseling process. Doing so would be impossible with a hand-guided tool, probably resulting in harm to the mason, but the robotically actuated chiseling tool developed for AROSU was capable of performing that task, creating elaborate, spatial surface patterns that were previously impossible to achieve.

To make that functionality accessible to the masons, an extensive range of Grasshopper components was created, representing the robotic cell, tool, and five stone structuring strategies for linear, parallel, V-shaped, bitmap-based, and 3D processing, as well as the algorithms for tool actuation, result pre-visualization and output. These components expose just the minimum of parameters, allowing masons to use Grasshopper as a simple user interface rather than a programming environment (Figure 44).

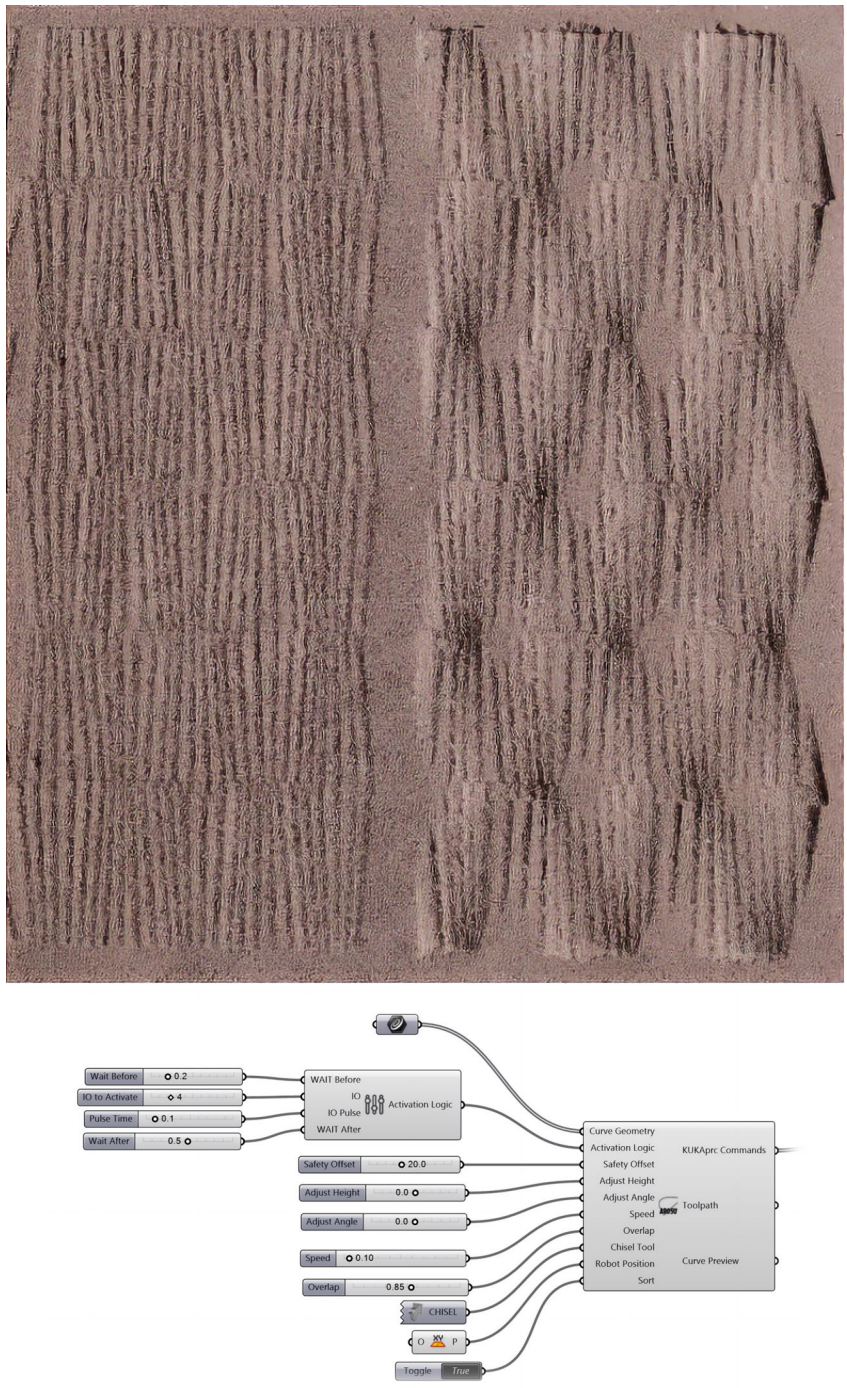

Figure 44: Novel stone surface patterns exposed through domain-specific components in visual programming.

The AROSU project demonstrated how careful analysis and collaboration with process experts are crucial for automating complex processes. Once the process data was known, visual programming enabled us to rapidly iterate results based on the craftsperson's feedback towards equalizing the manual and machinic chiseling results. Visual programming then also acted as an accessible user interface that exposed only the main features to non-programmers but allowed programmers to perform deep changes within the visual code as well.

Credits: The AROSU project was a collaborative effort with multiple partners. Sigrid Brell-Cokcan and the author were the primary researchers at the Association for Robots in Architecture, with the author being responsible for the programming strategies described above.

5.2.2 Absolut Spraypainting

The Absolut Spraypainting project (Braumann and Brell-Cokcan 2014) was developed for the beverage company Absolut for the occasion of the launch of a limited edition of their vodka Absolut Originality. Due to a unique manufacturing process with a swash of blue paint flowing along the bottles, each bottle had an individual look. The underlying concept was an activation towards promoting the idea of individualization. As such, the project had to:

- Come up with a way to provide deep individualization.
- Develop a visually appealing process with fast and elegant movements to attract an audience.
- Optimize the process to be highly reliable.

The initial concept proposed the spraypainting of white t-shirts with individual portraits – the core challenge of the project was not the path planning and toolpath layout but the development of the basic spraypainting process. In industry, robotic spraypainting is primarily explored within the context of evenly covering a given surface (H. Chen et al. 2002; Pichler et al. 2002) rather than creating specific aesthetic effects.

Absolut did not want an “industrial look, “so standard manual airbrush equipment was used. Initial experiments quickly showed insufficient process knowledge regarding the airbrush setup (nozzle, amount of paint, general handling, etc.) to achieve a good result. As such, a professional airbrush artist was hired to consult. The artist quickly optimized the choice of paints and airbrush; however, when his airbrush movement was recorded with a stereo camera in 3D, it became clear that his output depended on extremely fine control of the three parameters: distance, speed, and air pressure (equaling the amount of paint) (Figure 45). Duplicating that process would have required an actuated tool whose development was not feasible within the short timeframe of the project.

Figure 45: Manual prototype (left), robotically fabricated result with static air pressure, varying only distance and speed (right).

That made it necessary to develop a robotic airbrush strategy that builds upon the knowledge of the process expert while staying within the constraints of the robotic system and augmenting the process with the advantages of a robotic arm, such as high speed, accuracy, and full spatial awareness. At the same time, the design had to incorporate aspects of the Absolut Originality bottles with the blue swash of color.

The solution was to use a tool design with two separate airbrushes, one with black paint and a fine stroke and the other with blue paint and a broad stroke. A screw permanently sets their trigger, and only the air supply itself is controlled through the internal valves of a KUKA Agilus robot.

For the planning, the concept was to take a portrait, turn it into black-and-white, and then vectorize it to generate outlines. Parallel lines would then be laid across the outlines, showcasing the robot's accuracy and speed in a way that would be difficult to achieve by hand. Once the portrait was done, the second airbrush would apply parametrically individualized blue strokes to connect the t-shirt with the bottle design visually.

Technologically, this was achieved by first creating a 3D scan of the torso, where the t-shirts would be mounted. To locate the 3D scan within the robot's workspace, the robot was moved manually to several positions on the surface of the torso, and their XYZ coordinates were recorded. Using the evolutionary solver Galapagos, the sparse point cloud was matched with the mesh within Grasshopper, and then the toolpath layout of the toolpaths was changed to avoid collisions and singularities.

For the vectorization process, a custom component was developed for the visual programming environment that would use the computer vision framework OpenCV (Bradski 2000) to remove the blue-screen background, set a threshold for the black-and-white conversion, and then group clusters of black pixels intelligently as to achieve a maximum of visual recognition with a minimum of toolpath length, to optimize the output of t-shirts per day. Finally, the vectorized outlines are output by the component's output as a list of polylines.

As the installation was to be operated by non-expert staff, all the settings were exposed in a custom graphical user interface that was superimposed over the flow-based programming environment. As such, photos were loaded automatically from a camera to a wirelessly connected PC, so that the staff had to load the photo, adjust some key vectorization/posterization parameters to accommodate, e.g., different skin color, and then check the visualization within the 3D viewport, without having to directly interact with the parametric graph itself (Figure 46).

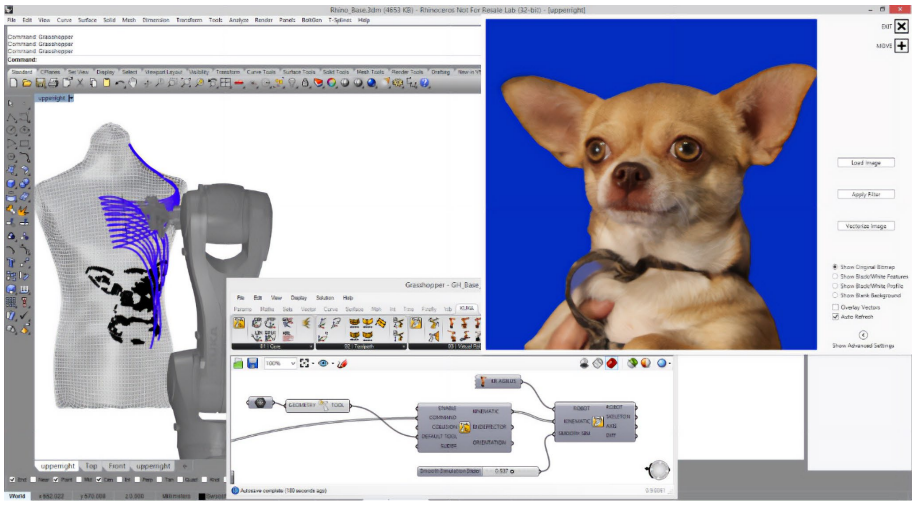

Figure 46: Screenshot showcasing the custom vectorization interface and process pre-visualization.

Grasshopper was essential to rapidly turn planar polylines into a complex, spatial, collision-free toolpath and to incorporate a high-fidelity visualization of the spraypainting output, showcasing the strengths of visual programming for both rapid process prototyping and expert-guided iteration, as well as a user-interface for non-expert users.

Credits: The project was developed by Sigrid Brell-Cokcan and the author. The author was responsible for the programming aspects presented above.

5.2.3 Polishing of Highly Accurate Automotive Parts

Puhl Oberflächentechnik is an SME based in Upper Austria with expert knowledge in postprocessing milled parts for the automotive industry, aiming to achieve a surface finish that would otherwise not be achievable solely through milling. In 2019, the company approached KUKA CEE and the author and presented a current project for which the company manually polished molds for automotive head-up displays to a mirror finish, being one of two companies worldwide with the expertise to fulfill the tolerance requirements of the automotive companies. The manual polishing process, for which a vibrating tool was equipped with a series of polishing pads of different sizes and grains, was very time-consuming, fostering the need to automate the process (Figure 47).

Figure 47: Manual polishing (left), developed robotic process (right).

While there are numerous polishing robotic applications in industry (Takeuchi, Asakawa, and Ge 1993), the company wanted to continue utilizing their current, certified process as they knew it could achieve the required tolerances. This led to the following challenges:

- Careful analysis of the manual process.
- Evaluation of a collaborative robot as a high-end polishing tool.
- Development of a path planning process that would create an arbitrary amount of step downs, with each layer having variations to avoid polishing the pattern into the material.
- Knowledge transfer to allow the process experts to program the robot by themselves.

As such, the goal was to implement that strategy with a robotic arm within a concise timeframe. Following a careful analysis of the manual process, the toolpath strategy was parametrically developed in Grasshopper, generating a continuous, helical movement over the free-formed surface of the base object (Braumann 2020). The central aspect of polishing is to remove the material as evenly as possible. Therefore, it is crucial to consider the size of the polishing pad as a factor for the toolpath design.

Puhl uses 3D printing to fabricate custom polishing pads depending on the object's curvature, material, and size. Therefore, the parametric

model makes it possible to implement new polishing pads in the code rapidly.

While accurate CAD files were available for every part that had to be polished, creating a robotic system that was accurate enough to apply constant pressure to the object was impossible. Depending on the offset, the object would receive more pressure in some parts and less pressure in others, which was unacceptable to the company. Therefore, it was necessary to implement sensors that correct for differences in pressure in real time.

Due to the short integration time, it was decided not to apply an external force sensor but to use a robotic system with integrated force-torque sensors, the cobot KUKA LBR iiwa, which had the added advantage of requiring less safety equipment than regular industrial robots. Rather than manually implementing a force-sensor via a fieldbus interface, the iiwa allows the user to set the force-related modes directly in code for each movement (see Chapter 4.7). In the current application, the iiwa robot would be put to a compliant mode with relatively high stiffness, meant to compensate for the inaccuracies of the calibration process, with an extra force that is applied along the tool axis towards achieving a constant pressure at the tooltip that can be parametrized in Newtons within KUKA|prc.

An initial polishing test with an LBR iiwa 7 R800 robot showed that the approach had merit but reproducibly stopped with a force-sensor-related error message after less than a minute. The mechanically stiffer LBR iiwa 14 R820, providing twice the payload, did not suffer from the same problem.

Using the iiwa reinforced the decision to use a custom polishing strategy using visual programming and KUKA|prc, as no commercial, general CNC system currently supports code output for Sunrise iiwa and its force-sensitive movement modes.

An external evaluation of Puhl's client showed the results to be within specifications at a roughness Rz1max of 0.254 µm compared to the 2.595 µm after high-precision milling (Figure 48). Puhl directly interacts with the Grasshopper definition for programming by setting the relevant core parameters and then duplicating and/or individualizing the jobs

according to the craft knowledge. Individualization is essential as multiple, identical polishing processes would leave a pattern on the metal, making the polished piece worthless. KUKA|prc accepts the settings and then exports an XML file with the toolpath that the iiwa's operator can easily load without requiring Grasshopper knowledge.

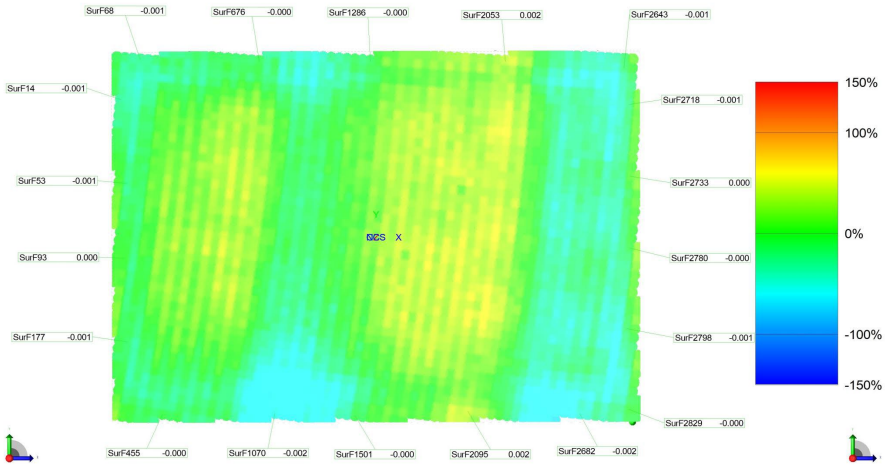

Figure 48: Surface finish measured on a Mitutoyo Strato APEX 776 coordinate measurement machine.

Credits: The polishing project was developed and programmed by the author.

5.2.4 Robotic Saddlemaking

In industry, where serial production happens in high volumes, the improved efficiency of robots means that the number of robots scales with the production line's throughput. If a robot can perform the jobs of three workers, then two robots would replace six workers on the line. At small enterprises, the speed and efficiency of a single robot may already exceed the speed of another labor-intensive process, causing the robot to have downtime while the rest of the workflow catches up with its output.

As such, it is even more critical than in industry to carefully analyze the entire production chain to identify processes that may be automized so

that the output increases, rather than just shifting bottlenecks at a high development cost. A complete lights-out automation is hardly ever financially viable for small enterprises, as the product's value is mainly derived from being handmade locally rather than imported from low-wage countries.

The saddle maker Niedersüß in Rohrbach, Upper Austria, is a small enterprise with less than 30 employees that creates high-end riding saddles for customers like the Spanish Riding School in Vienna, selling half their products nationally and exporting the other half, mainly to countries like the USA and Japan. As a family-owned company founded 300 years ago, Niedersüß has a long tradition but was highly interested in supplementing their craftspersons with new technology. Toward that goal, the challenges were the following:

- Analyze a craft process towards finding potential for automation.
- Propose ways in which individualization and automation may improve the production process.
- Provide knowledge transfer to enable the craftspersons to control such a process.

Niedersüß was already molding saddle cores with their polyurethane molding machine: These components were previously chiseled out of wood by carpenters, a traditional process that is no longer financially viable today.

As an initial step towards introducing robotic processes, a thorough analysis of the entire fabrication process was performed (Figure 49), tracking the total time it takes to fabricate a saddle and each process's potential for automation (Braumann 2020). While, for example, subtractive processes are highly suitable for automation, the sewing process would be very challenging, as the leather needs to be stretched over the core without folding in a particular way during sewing.

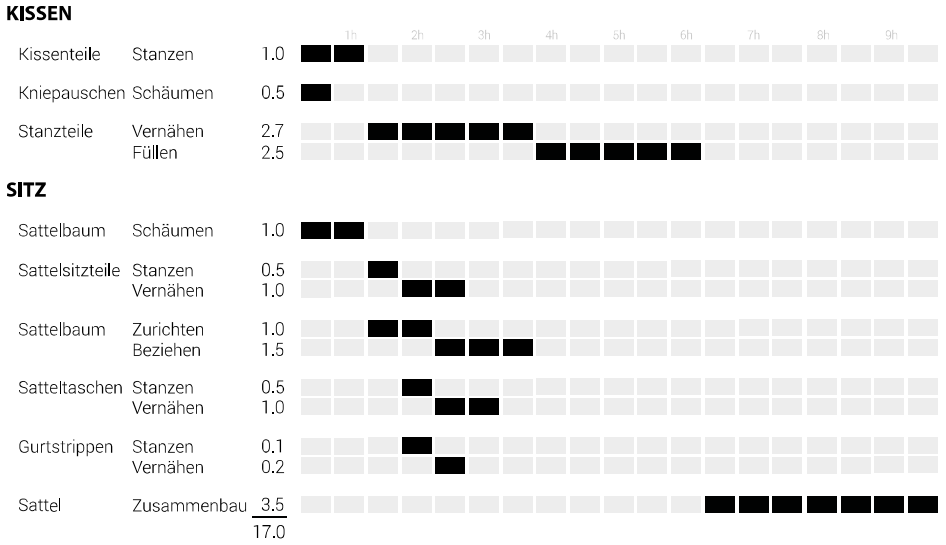

Figure 49: Process analysis of the saddlemaking workflow to evaluate the potential for automation.

Two processes were rated highest regarding the ratio between effort and gain of automation: The saddle cores' production and the leather's cutting. For each application, our research also identified a potential value of the robotic process over the current fabrication. For the saddle cores, a new mold must be milled out of metal for each new form. As such, similar to the commercial fashion industry, saddles are only offered in a few standard sizes and then need to be adapted to the individual rider and horse by selectively adding and removing foam padding. Instead, the analysis suggested fabricating a larger offset form and then selectively postprocessing it with a milling cycle, cutting it precisely to the individual specifications (Figure 50). The cutting of leather was done with a stamping process, requiring the craftsperson to bend and sharpen a metal form for each piece of leather so it could be punched with a heavy-duty press. Instead, a robotic cutting process with an oscillating knife would greatly facilitate individual designs and customization.

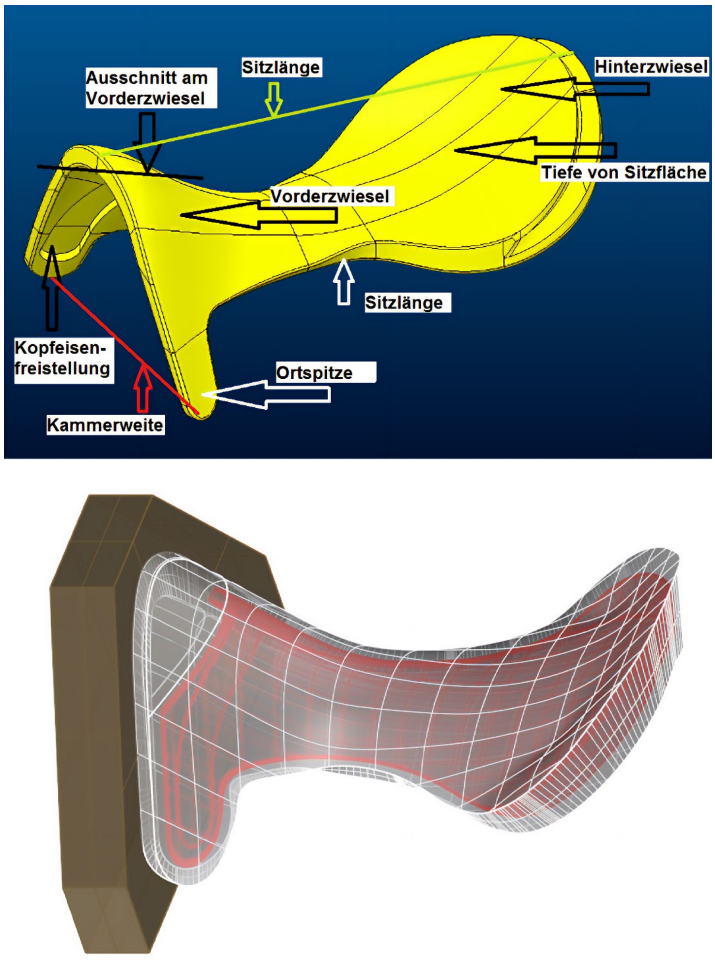

Figure 50: Analysis of relevant saddle parameters (top), offset model as the basis for individualization (bottom).

Individually, each process would have exceeded the capacities of the subsequent production steps. However, the multifunctionality of robotic arms now allowed us to propose a split fabrication, with the robot splitting its time between the fabrication of saddle cores and leather elements.

For the parametric production of the saddle cores, a prototype was developed at Creative Robotics, introducing the craftspersons at Niedersüß to visual programming and KUKA|prc. This has led the company to purchase their own robot and implement it as a design tool,

influencing the design and fabrication of new saddles to facilitate robotic fabrication while still relying on manual labor for the assembly. Programming is done via KUKA|prc for custom processes and combining Autodesk Fusion 360 and KUKA|prc for the multi-axis CNC milling of prototypes (Figure 51).

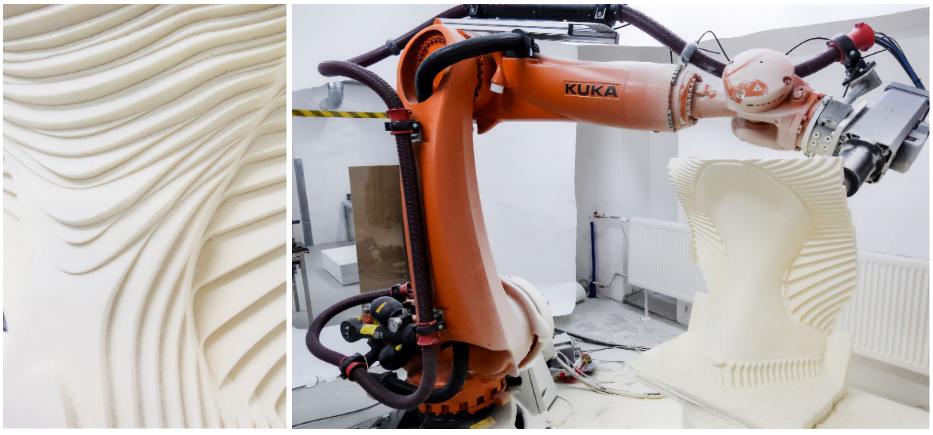

Figure 51: Prototype milled via KUKA|prc and Fusion 360.

Credits: The project was developed and programmed by the author. Sander Hofstee and Benjamin Greimel contributed to the analysis and visualization.

5.2.5 Additive Sewing

The startup Yokai has identified the textile industry as an area with great potential for digitization, with consumers' increasing awareness of the working conditions in low-wage countries and, thus, rising demand for locally produced textiles. However, the high labor costs in Austria restrict handmade textiles to very high-end fashion, so larger-scale production depends on automation. Currently, industry is developing highly complicated machines to replace manual production lines, mostly directly replicating manual processes.

Yokai instead aims to re-think existing processes, bringing textile fabrication from 2D into 3D via simulation and robotic fabrication. A core aspect is the connection of textiles, which is most commonly done via sewing. However, sewing for more complex textiles is challenging to

automate (Molfino et al. 2008). Instead, Yokai proposed using an extrusion process to weld textiles with flexible, spatial seams. A prototype was developed in collaboration with the author, repurposing existing commercial extruders to connect textiles spatially. This led to the following challenges:

- Developing a way of precisely controlling an extruder through a robot.
- Setting up a software workflow to intelligently inform the production process.

Figure 52: Connection and decoration of textiles via additive manufacturing technologies. From prototype (left) to automated production of face masks during the COVID-19 pandemic (right, Yokai Studios).

Visual programming and KUKA|prc allowed the development of custom fabrication strategies, generating, e.g., 3D zig-zag lines along 3D scanned objects to connect textile sheets. An essential aspect of that is the fine-grained manipulation of the density of connection elements depending on the projected stress of the textile. This density can be modulated geometrically, e.g., by increasing the frequency of the zig-zag toolpath and physically by increasing the amount of extruded material.

The need for being able to manipulate the extrusion flow also arises from the robot's kinematics: Where a three-axis kinematic system – as used

for most 3D printers – can achieve very constant speeds, a robot's more complex kinematic system can, e.g., move past singularities that significantly decrease the speed of a tool center point. Compensating slowdowns is, therefore, crucial for robotic applications.

Ideally, an extruder would be driven with a KUKA motor as an external axis so its position is perfectly synchronized with the toolpath. However, small-size KUKA controllers do not provide sufficient space for the electronics to drive external axes. As such, a PLC-based system was developed, built around a Beckhoff PLC with interface cards for stepper control for driving the extruder, a digital output connected to a solid state relay to activate the heating element, and a PT100 input for reading its temperature. The KUKA Agilus-2 robot is configured with a virtual – i.e., simulated, not physically existing – external axis. It transfers its axis value in degrees via an EtherCAT bridge in real-time to the PLC.

On the PLC, the axis value is connected to a virtual encoder, setting the position of a virtual axis to the position of the KUKA's simulated axis. To drive the extruder's stepper motor, the extruder is coupled with a virtual gear system to the virtual axis. This results in a system that can drive a non-KUKA motor comparable to a regular external axis.

To complement the mechanical implementation, KUKA|prc and Grasshopper are crucial for software control, as the developed process goes beyond the state-of-the-art in CNC software. The value of the external axis is set in relation to the point distance so that, e.g., 5mm of toolpath is equivalent to 2mm filament, which needs to be converted into degrees by considering the gearing of the extruder and the diameter of the feeding wheel. This value can be overridden or changed to accommodate the requirement for thinner or thicker seams. Grasshopper thus allows the easy postprocessing of external axis data to fit the developed mechanical system.

The Yokai approach combines existing technologies from 3D printing and industrial control systems into a completely new application that depends strongly on the capabilities of visual programming to define extrusion speeds in relation to geometric and process-related constraints, creating a robot-centric, highly efficient way of connecting textiles.

Credits: The Yokai textile concept was developed by Michael Wieser and Viktor Weichselbaumer. The PLC and visual programming implementations were developed and programmed by the author.

5.3 User-Centered Robotic Interfaces

To make industrial robots accessible to new users, it is necessary to provide domain-specific interfaces that expose the relevant parameters in an accessible format to the user. The depth of that customization and level of abstractions depends on the level of the user, ranging from providing APIs to minimalistic user interfaces that provide only the most basic number of parameters, up to gestural interaction (Braumann and Brell-Cokcan 2012). KUKA|prc's flexible architecture supports either of those approaches.

5.3.1 Timber Manufacturing

ZÜBLIN Timber GmbH is a timber manufacturer based in Germany. Züblin has been one of the first users of robotic fabrication in architecture and construction, acquiring a KUKA robot in the early 2000s. The challenge encountered by Züblin has been that CAM software is generally optimized for metal processing, being the by far largest market, while timber software targets specific joinery machines. As such, the engineers at Züblin used the programming environment that was best known to them and created a robot postprocessing code within Microsoft Excel through Visual Basic. This software has been successfully used for large-scale projects like the Elefantenhaus in Zurich (*Mikado* 2014). However, it did not provide any simulation capabilities. Initially, it was built based on the users' initial knowledge state, not using KUKA standards like tools and base numbers but dealing with those complexities through custom-developed algorithms. As a result, the software became hard to maintain and depended on a single engineer for maintenance and expansion. Züblin then approached the Association for Robots in Architecture and expressed an interest in KUKA|prc and integrating their large-scale robotic setup with two external axes in the software.

A key goal in implementing KUKA|prc at Züblin was knowledge transfer so that all robotic processes could be handled in-house, avoiding the legal challenges of sharing confidential data with outside companies. To do so, Züblin hired a student to receive KUKA|prc training; he is today in charge of robotic processes at Züblin.

By integrating KUKA|prc with Grasshopper, Züblin can now quickly incorporate algorithms developed either in-house or by its geometry consulting subcontractors into a single environment. Instead of adapting milling strategies developed for metal fabrication, the company now develops its own process- and machine-specific strategies, creating valuable intellect property and increasing its independence.

Figure 53: Parametric robot control at Züblin Timber (Züblin Timber).

For Züblin, KUKA|prc acts as an architecture-specific user interface for developing custom timber fabrication strategies and has since been used for large projects like concrete free-form formwork for the Stuttgart 21 train station project.

5.3.2 Robotic Guitar Playing

The Robotic Guitar Playing project aimed to develop a robotic system capable of playing a guitar. The challenges were:

- Getting a deeper understanding of the process.
- Developing a strategy that is optimized to the strengths of the industrial robots that are used.
- Creating an interface allowing a non-programmer with extensive process knowledge to program a performance intuitively.

While Grasshopper provides an accessible interface that users can quickly grasp, data entry can only happen locally through numeric/text input or referencing external data sources, e.g., importing geometry from Rhinoceros 3D. The Robotic Guitar Playing project explores how geometry can be interpreted and parsed in new ways towards augmenting toolpaths with information regarding speed and timing.

A creative agency approached the author with the idea of having a guitar playoff between an industrial robot and a speaker at their upcoming event in Geneva, Switzerland. This was inspired by the Automatica music video by Nigel Stanford (2017). However, while the music video is heavily edited and augmented with special effects, the goal was to have a robot play the guitar in real-time.

The first step towards realizing the project was analyzing the guitar-playing process, particularly the involved speeds. It quickly became apparent that speed was a primary concern, as the average wrist movement can easily reach 0.5 m/sec (Vaisman, Dipietro, and Krebs 2013), while one of the fastest KUKA robots, the KR3R540, reaches at most 0.78m/sec (at 600 degrees/second and a 75mm radius) at its peak for a wrist motion. While fully articulated, 5-fingered robotic hands are available commercially, they are not optimized for speed and are complicated to control. Therefore, the robot setup was defined to consist of two high-speed KR3R540 robots, as it seemed unlikely that three or more machines could share a guitar's limited workspace.

The next step was to analyze how a maximum variance can be achieved for guitar playing with only two fingers, i.e., a fifth of what guitars are optimized for. Sander Hofstee, a semi-professional guitar player consulting the project, did this entirely manually, without any digital tools.

Based on his results, two robotic endeffectors were developed, one holding a plectrum for plucking the strings and the other one with two dampening tools, one for dampening a single string and the other one for dampening all strings, which could be switched by changing the tool orientation (Figure 54).

Figure 54: Guitar playing simulation (top) and prototypes with 3D-printed endeffectors (bottom).

Programming two robots to play the guitar brings along two challenges: Synchronization and rhythm, which are closely related. In a technical sense, KUKA offers the RoboTeam tech package for collaborative processes involving multiple robots, offering the option to either start and stop movements at the same time, to have a robot wait for another robot, or to have one robot constrained to the other robot's tool coordinate system.

A synchronous movement is not helpful because the speed for using a plectrum and dampening a string is widely different. Instead, the movements need to be triggered at the same time. To avoid the complexity of extra software, this was realized not via RoboTeam but by connecting the robots in a star-topology via EtherCAT master-master couplers.

In KUKA|prc, the synchronization capability can be switched between RoboTeam and the generic solution used here. A variable for input data and one for output data is defined and mapped to the bits transferred by the EtherCAT coupler. The user can set the size; unsigned 16-bit integers were used in this case, allowing up to 65536 synchronization steps.

At the location in the KRL code, corresponding to the synchronization component in the visual programming data flow, the variable is iterated by one and then waits for the value of the connected robot to reach the identical value. Once both input and output variables match, the robot continues its movement.

Rhythm and the programming of the movement is a much bigger problem. While, e.g., wait times can be set in KUKA|prc to achieve specific pacing, the time calculation of the simulation environment is not sufficiently accurate to predict the exact timing of the robot. Therefore, timing and rhythm require an extremely iterative workflow with a proficient person listening to the results and adjusting the time values until the right rhythm is achieved. Doing this numerically proved to be very slow and unintuitive, also reflected in the results.

To allow the guitar player to program the complex guitar-playing movements, a new process was developed that is closely related to the concept of notation, where the distance in X is associated with the time, and the shape and Y position of the note defines the sound. To realize

that in a CAD environment, the viewport may be considered the front end where the user interacts with notation, while the translation into robot movements and wait commands is happening within the visual programming "backend," with which the user himself is not interacting.

A series of core guitar movements and their parameters, such as speed, are identified toward that goal. Each guitar movement is represented by a single polygonal geometry with the centroid as a clearly defined reference point. Modifying the position, layer assignment, size, and number of sides of the geometry sets parameters for guitar movements, changing the speed, intensity, number of strings plucked, the dampening, or other parameters.

The Y-axis of the CAD drawing assigns the starting string, while the X-axis defines the speed: The longer the distance between two motions, the longer the wait times are inserted. This programming process is done for both the robot with the plectrum and the dampening tool, with the synchronization represented by lines that connect both timelines (Figure 55).

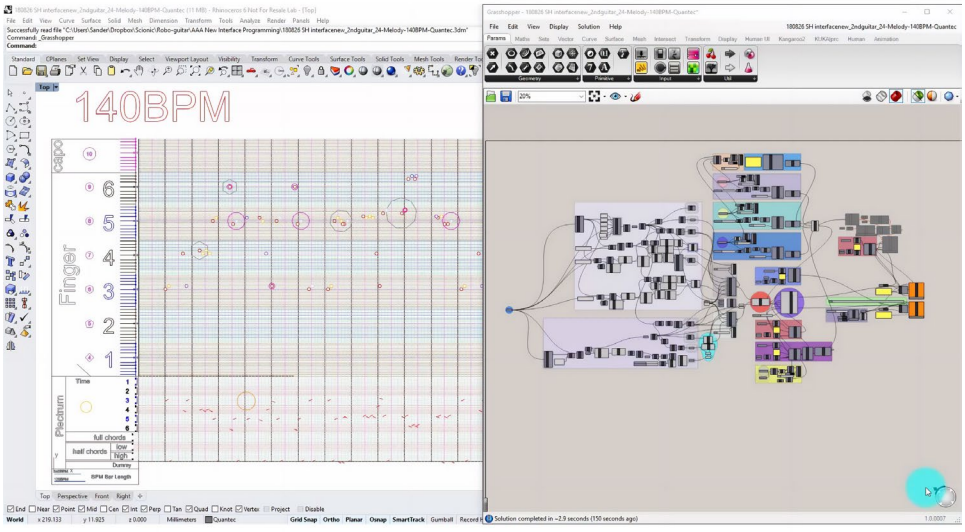

Figure 55: KUKA|prc allowing a musician to program two collaborating robots with a custom notation system.

The advantage of that process is that speeds and timing can be rapidly changed by simply moving geometry around, changing the layers, or

even transforming the notation geometry. Changes to the code result in a recompute within the visual programming environment and KUKA|prc, immediately outputting a new simulation as well as copying the two resulting SRC files to the robots via Ethernet, enabling a rapid iteration and optimization while keeping very close to the musician's expertise in notation.

Ultimately, to properly showcase the complex setup in the large event location with over 6000 viewers, a third, heavy-duty Quantec robot was added to the project. While the large robot did not interact with the guitar itself, it moved the entire setup in a way inspired by a human guitar player moving on the stage (Figure 56).

Figure 56: Robotic guitar performance at Geneva Fair.

Credits: Concept and programming by the author, guitar choreography, and consulting by Sander Hofstee.

5.3.3 Custom User-Interfaces

Custom user interfaces (UI) allow even novice users to interact with robotic processes without requiring any interaction with code or (visual) programming (Braumann and Brell-Cokcan 2014) as they greatly abstract a robot process to a very minimal amount of input data, such as a drawing. Depending on the project's scope, user interfaces can be implemented on top of a flow-based programming environment or even as the functionality of KUKA|prc as a standalone or cloud environment. The challenges are:

- Communicating between the UI and the robot programming environment.
- Providing a meaningful and custom-tailored interaction for the end-user.

An example of a superimposed UI was developed for KUKA CEE for a robotic application at an event around Halloween. While the initial idea was to cut pumpkins, safety issues prompted a switch to a safer drawing process. The desired workflow is that the customer would draw a custom shape on a tablet, with the robot then drawing the 2D stroke onto the physical pumpkin with a pen (Figure 57).

Figure 57: Workflow to apply a 2D pattern onto a free-formed surface.

The interaction between the user and the pen is handled via a Microsoft Surface tablet and its included pen. Due to the optimized support for pen input, the drawing UI was developed as a Windows Universal app, enabling very aesthetic, pressure-sensitive strokes on the canvas. After completing the design, the `InkStrokeRenderingSegment` strokes are translated into Bezier segments and then saved as a TXT file on a network share located on an external PC.

Grasshopper monitors the file on that PC: Once its content changes, new robot toolpaths are generated and projected spatially on the previously 3D scanned pumpkin geometry. After checking for reachability and

collisions, the toolpath is immediately streamed via mxAutomation to a connected KUKA robot, drawing the input design in 3D.

Implementing user interfaces through standalone applications results in task-specific, optimized platforms that do not require a license for CAD software. A prototypical implementation has been developed within the game engine Unity (Chapter 4.3, Figure 58), allowing the user to record positions by intuitively moving a marker in 3D space and saving its position.

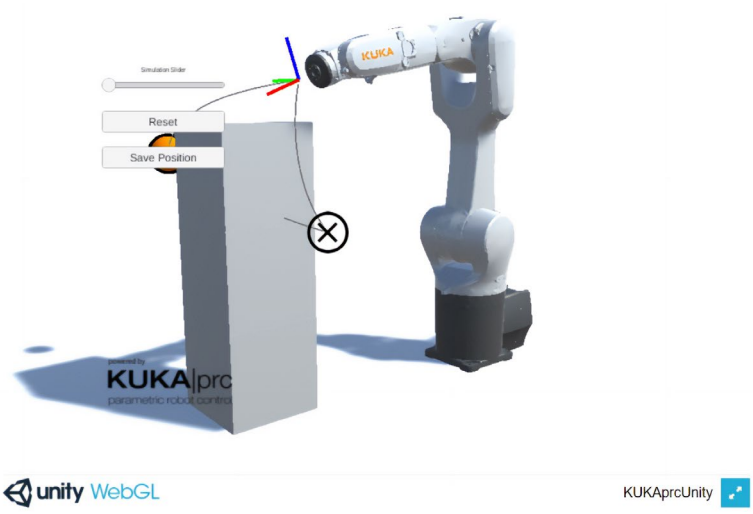

Figure 58: Robot control through the browser using Unity and WebGL.

The robot simulation is coupled with the gaming engine's physics solver, accurately simulating, e.g., box collisions. An advantage of an environment like Unity is that various other technologies, e.g., for mixed reality (see also Chapter 6.2.2), are already implemented and can, therefore, be rapidly connected with robotic processes.

With the increasingly maturing WebAssembly technology, allowing C# - and other languages - to run within the browser through a portable binary-code format, these applications can even be run directly within the browser and on mobile devices, significantly increasing the potential range of applications.

Credits: Pumpkin UI realized with Karl Singline, other aspects by the author.

5.3.4 UI in the Cloud

Running your own software in the cloud provides complete control over the software and its use. However, there are challenges to achieving this goal.:

- Setting up a scalable infrastructure for robot control.
- Providing an accessible 3D interface for interacting with the robot.
- Coming up with approaches towards managing the near-infinite scaling of cloud systems despite a limited number of robotic production machines.

While anyone can set up a server, it is hard for non-experts to set up a scalable infrastructure, e.g., for robot simulation in the cloud. Therefore, a prototype was realized through Azure Functions, a scalable way of serverlessly running code. To bring KUKA|prc to the cloud, wrapping the core KUKA|prc functions in a REST-API was necessary. Each method is represented as an IActionResult, triggered by a HttpTrigger, usually a GET request.

This was implemented as a prototypical application for Print-a-Drink, a custom robotic fabrication process developed by Benjamin Greimel that deposits microliter drops of oil within a cocktail, thus creating patterns and logos. The development of Print-a-Drink has led to the founding of a startup, which has since collaborated with companies such as HP Enterprise, Schott, and Voest and is closely collaborating with the US-based whiskey brand Bulleit.

In general, patterns for 3D printing are prepared in advance, only allowing the user to select between different designs. The idea of the prototype is to tackle the challenge of scalability and efficiency in mass customization. The most straightforward approach would have been to provide drawing stations for each robot, as for the pumpkin project. However, doing so would significantly reduce the number of drinks printed per hour. On the other hand, creating a design on a mobile device is also feasible. However, if all drinks are queued up, it becomes hard for the users to identify their drink.

The developed, cloud-based process is inspired by printing workflows in larger companies, where printing jobs are sent to a printer and then need to be manually unlocked. The Autodesk Forge cloud platform was used to develop a web interface that allows the user to define the positions of the points within the glass in 3D by rotating a construction plane around the glass's center to face the camera (Figure 59). Once the design is finished, it is possible to simulate it: KUKA|prc running on Azure Functions – a serverless compute platform – accepts the points, creates a toolpath, and then sends back the transformations for each position with the previously set interpolation. A robot position requires at least seven transformations (base plus six axes), and a transformation may be represented by a 4x4 matrix (16 values with 32bit); at least 4kB are transferred per position. A 500mm long toolpath with a 2mm interpolation distance results in one Megabyte of data, a manageable amount of data.

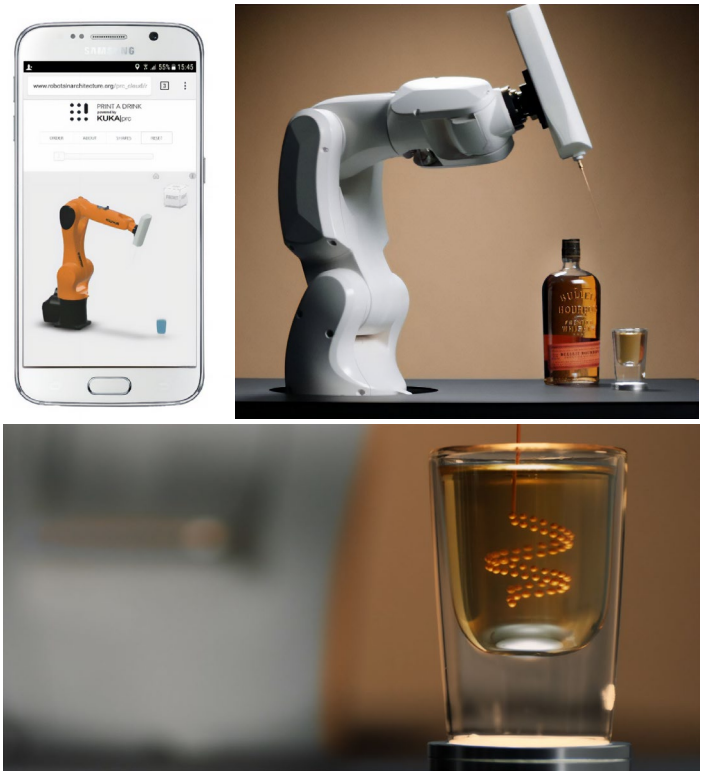

Figure 59: Print-a-Drink mobile user interface prototype (top, left), robot setup (top, right), detail (below, Print-a-Drink).

When a user uploads data, it is transferred to an Azure Table database and returns a QR code containing the internal path to the data object. With that QR code, the user can now queue up at the robot and scan the QR code once it is their turn. A custom component in Grasshopper interfaces with OpenCV to read the QR code retrieves the data from Azure, checks it for plausibility and safety, and then streams it via mxAutomation to the connected robot, printing the individualized drink.

The advantage of running KUKA|prc directly in the cloud is that there are no different software versions and that IP is more securely protected. Serverless computing strategies allow the processes to scale, dynamically serving a few the same as a few hundred parallel users. However, the returned simulation data is comparably extensive, requiring a fast internet connection. A combination of local WebAssembly and cloud-based computing may provide the best features for browser-based applications.

5.4 Feedback-Based Processes

A simulated robotic process always depends on the assumption that the physical and digital environments match within a small tolerance. Feedback-based processes through KUKA|prc allow users to define general fabrication workflows and strategies and then adapt them to the individual use case in real-time via captured sensor data. A specific challenge is dataflows that break from the acyclic graph model generally used for visual programming.

5.4.1 Rich Data for Safety

The Responsive Safety project is based on a collaboration with the computer vision startup AMB, which developed a person-tracking algorithm, making the position of a person and its hands accessible through a REST API. The concept of the project is to investigate how future safety systems, e.g., safety-certified 3D cameras, may change current safety concepts based on a Boolean safe/not safe.

Rich data providing, e.g., the position of people, may be used to adjust robotic processes to optimize cycle time and robot safety. Currently,

only a few optical, 3D safety systems, such as the Pilz Safety Eye, used, e.g., by Jalba et al. (2017), are available on the market and generally only output whether a designated area is safe or not, without any further metadata. This prototype explores the following:

- Interfacing with a new vision technology.
- Development of a robotic system that reacts to external input towards reducing the risk of humans within its workspace.

The prototype detects persons using AMB's proprietary algorithms (Figure 60). A custom component developed for Grasshopper cyclically accesses that data and makes it available within the visual programming environment. Initially, a large number (>50) of pick-and-place jobs are created within Grasshopper, with each list of movement commands grouped into a single data item, referred to as a Task.

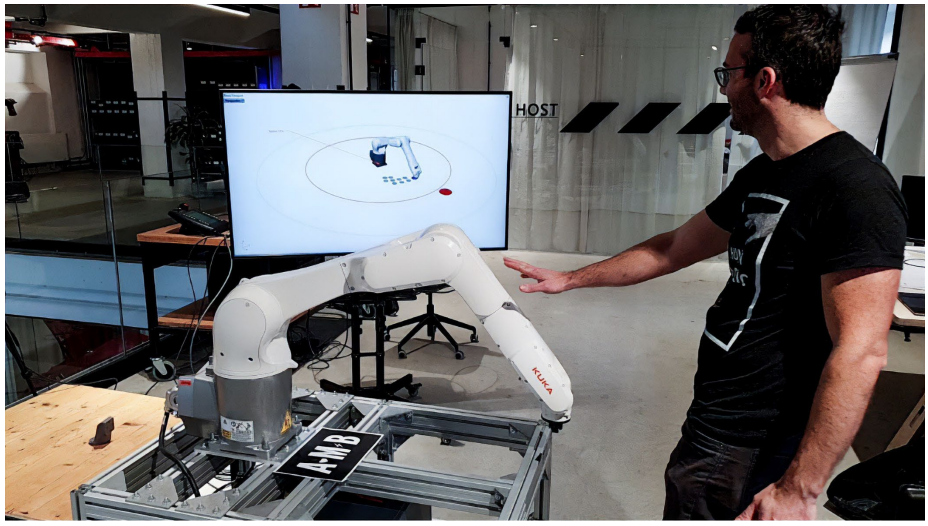

Figure 60: Software recognizing a hand input and robot reacting accordingly.

The robot setup uses mxAutomation in the Iterative mode (see Chapter 4.6), finishing jobs before accepting new data. As the output, mxAutomation provides the current axis values of the robot and its current state, whether it is processing commands or not. Grasshopper now queries the position of both the user and the robot. If the robot is not busy, the task furthest away from the user is chosen for processing to keep the robot away from the user.

In parallel, the *Core* KUKA|prc component is used to compute the 3D model of the robot based on mxAutomation's axis values. This enables the calculation of the closest distance of the user and the mesh of the robot. Based on the current distance, as well as the known future positions of the toolpath, the override speed of the robot is dynamically reduced from 100% down to 0% until finally, the `$MOVE_ENABLE` signal is disabled, stopping the robot.

While such a process would need to be supported by regular, certified safety equipment at the moment, it showcases the potential of future developments in that area and is the subject of an ongoing research project. The core logic of the prototypical implementation was developed within a day of preparation and on-site day, showcasing the capabilities for rapid process prototyping afforded by visual programming and KUKA|prc.

5.4.2 Creative Bin-Picking

While the previous project deals with constantly refreshed data, the Creative Bin-Picking prototype interfaces a visual programming environment with a high-accuracy Photoneo camera that needs to be manually triggered. The applied Photoneo PhoXi is a 3D camera with built-in object recognition through an integrated NVidia GPU. A 3D model of the object is loaded into the software to recognize the object, and its reference point is defined. The configuration file is then parametrized to set the detection parameters, e.g., whether the detection process continues until some elements are detected or until a timeout condition is met.

When the custom-developed Grasshopper component triggers the detection process, referencing the configuration file, it returns a list of transformation matrices, defining the position and orientation of the object's reference point and, thus, its spatial position.

The challenge of that project is:

- Integrating an individually triggered process into a continuous dataflow.

- Defining an iterative production process through a flow-based visual programming environment.

With the known position from the matrices, it is possible to define a pick-and-place process to manipulate the detected object (Figure 61). The challenge lies in the next cycle, as the robot has to move away from the working area, trigger the object detection again, and then perform the next cycle.

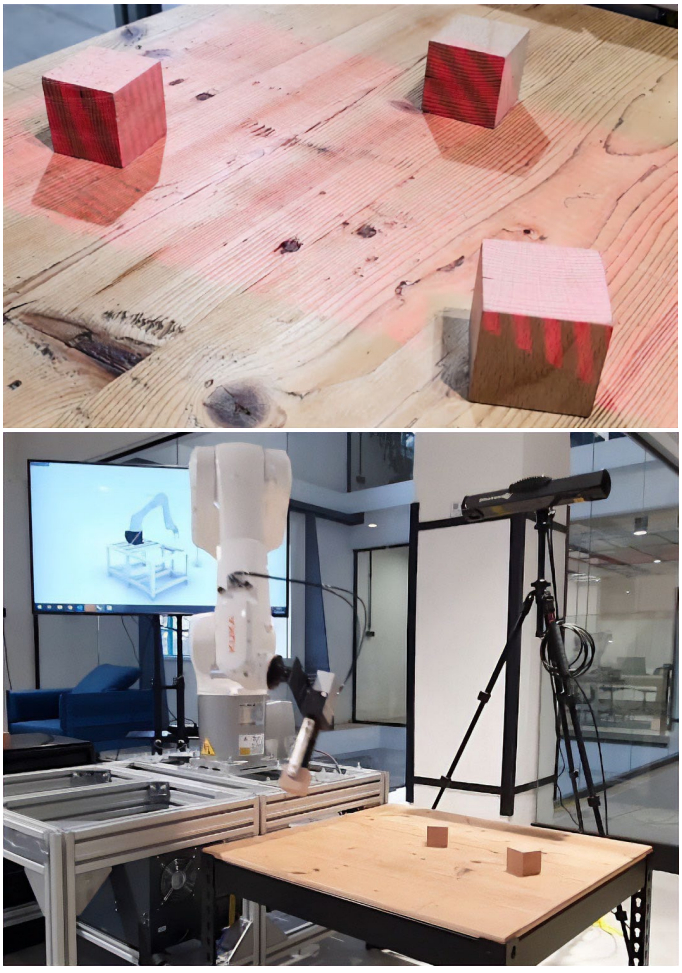

Figure 61: Object detection with a Photoneo scanner, manipulation using mxAutomation.

Two different approaches are developed to implement such a process within the dataflow constraints of a visual programming environment: Activation via outputs or spatial triggers. For the activation via outputs, the mxAutomation method `KRC_READDIGITALOUTPUT` is set up to query the state of a given digital output cyclically – while commonly, digital outputs are only set but not queried. As part of the toolpath, a *Set Digital Output* component is added at the end of the list.

Once the pick-and-place process is completed, the robot moves away from the workspace and sets the defined digital output. The *Feedback* component – decoupled from the dataflow – now queries the digital output and sets its state as a Boolean output connected to the *Photoneo* component's input, which starts the detection process.

As the detection process takes a few seconds, it is implemented as an asynchronous component. The trigger starts the process, and the component refreshes once the data is available. Due to how Grasshopper works, any data refresh, even if the data stays identical, causes all downstream components to refresh. As the output state is queried cyclically, the refresh rate depends on the cycle time but is generally around 5Hz.

The component buffers the previously captured positions and outputs them again to counter that. As the Cartesian position is identical, the generated hash is identical, and KUKA|prc will safely ignore the updated data.

An alternative approach would be to use spatial triggers, i.e., querying the distance between the robot, usually its TCP, and a given position in space. A Boolean value switches its state once the distance is below a certain threshold.

The cyclical refresh is also challenging for data persistence, as for a more complicated pick-and-place process, keeping track of the deposited elements will be essential. A similar hashing process had to be implemented to ensure that only one new geometry is added at every pick-and-place cycle rather than a new geometry at every robot visualization cycle.

These challenges clearly show that "geometric" flow-based visual programming environments are not optimized for real-time processing but rather directed dataflows (see Chapter 6.2). Doing so requires users and developers to re-think many software behaviors and counter them with specific workarounds and strategies.

Credits: Process logic by the author, pick-and-place implementation by Karl Singline.

6 Conclusion and Outlook

6.1 Revisiting the Research Questions

Chapter 1.1 presented three major research questions that guided the initial investigation into robotic processes within the creative industry:

- Is there merit in constraining the toolpath of a robot to parametric geometry through flow-based visual programming?
- Can programming and simulating a robotic arm be made accessible to the creative industry by implementing it within a visual programming environment?
- How can robotics advance and benefit the creative industries, resulting in entirely new approaches, products, and innovation?

It is challenging to answer these questions objectively, as the creative industries are a highly heterogeneous field. Therefore, the answers are given from the author's point of view at the intersection of architecture, design, crafts, and performative arts.

While architects have used early flow-based programming tools like Generative Components (Chapter 2.3) for generating G-code and supporting complex fabrication processes, the author's KUKA|prc software marks the first implementation of a complicated machine directly within a geometry-centric, accessible programming environment. Such an implementation may not have been necessary for machines with fewer degrees of freedom; however, having a native, immediate simulation of the complex robot kinematics (Chapter 3.3) proved to be a significant advantage for designers and craftspersons (Chapter 5).

The main factor in making the process of programming and simulating a robotic arm accessible to new users is its native implementation into a leading flow-based visual programming environment, i.e., an application-agnostic environment that many designers are already actively using but augmented with application-specific capabilities.

By custom-tailoring the software architecture and design philosophies (see Chapters 3.4 and 3.5) of the developed robotics software to fit the

standards of that environment, users can apply their existing knowledge for new tasks with little effort.

The work done within the context of Skill Digitization (Chapter 5) demonstrates the great potential of using robotic fabrication for the creative industry: Creative users do not need to develop and build new machines for non-standard processes but can use a generic robotic arm and equip it with the specific tool to perform a given task.

The past years have seen a constant increase in the creative industry's interest in robotics, as demonstrated by factors such as the increasing membership at the Association for Robots in Architecture and the number of attendees of the ROB|ARCH conference series. Thus, the growing number of users is vital in disseminating knowledge between peers, fostering innovation, and, in general, advancing digitization in creative fields.

6.2 Outlook

While the current state of KUKA|prc fulfills many of its users' requirements, the creative robotics field is constantly evolving. Several intensive future research fields exist within the context of KUKA|prc.

6.2.1 Cyclical / Asynchronous Software Architecture

The current software architecture of the developed software has been directly influenced by the software architecture of flow-based visual programming environments (Chapter 4). However, breaking the established norms of the host software, e.g., implementing new functionality such as cyclical data flows for interactive production processes, negatively impacts the user experience.

Therefore, it may be necessary to engage in a discussion with the developers of these environments to determine how to better represent cyclical data flows. Challenging applications beyond robotics are modern web APIs, which generally use asynchronous methods. Advances are being made in that field through interfaces such as Rhino.Compute.

Such an asynchronous workflow will require buffering and managing incoming data, as is currently done by KUKA|prc, but in a standardized way at the level of the flow-based programming environment rather than its plugins.

6.2.2 Mixed Reality

Already now, virtual reality is being used to plan and evaluate industrial facilities (Iqbal and Hashmi 2001). For the creative industry, augmented and mixed reality (Figure 62) may provide more immediate benefits, as robots and people in the creative industry more commonly share a workspace, compared to industry's "best-case" of lights-out manufacturing.

In the future, KUKA|prc may stream process information, like the robot's trajectory within the near future, to a mixed reality headset to increase the user's safety and visualize sensor data overlaid on top of the physical world. In turn, feedback such as the user's position, gestures, and gaze may be streamed back to the robot, going far beyond what is currently done, e.g., via projection mapping (Hietanen et al. 2019).

A leading platform for mixed reality is the game engine Unity, which already supports robot control via KUKA|prc (Chapter 4.5.4).

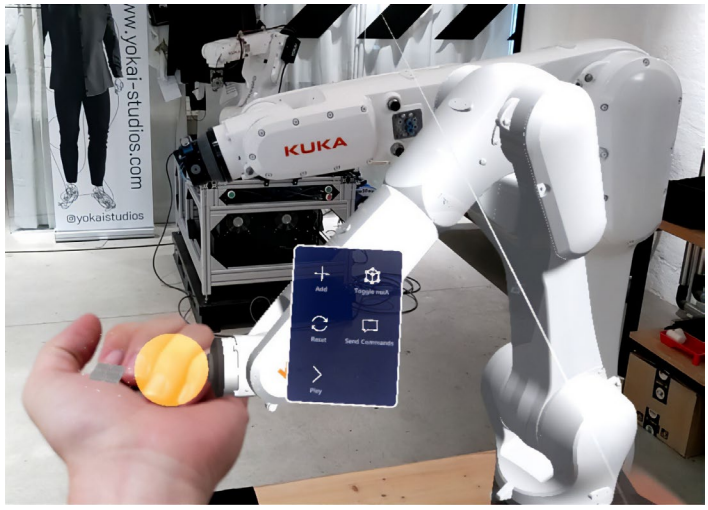

Figure 62: KUKA|prc running in a Microsoft Hololens 2 using Unity and the Microsoft Mixed Reality Toolkit.

6.2.3 Machine Learning

Machine learning is a significant area of investigation within the greater field of robotics, from object detection (Zhao et al. 2019) to grasp analysis (Caldera, Rassau, and Chai 2018) and predictive maintenance (Carvalho et al. 2019). In creative industries where mathematical and statistical knowledge is less prevalent than in engineering, automated machine learning frameworks (Zöller and Huber 2021) may open up machine learning as a new way of optimizing robotic processes.

Software environments such as Unity (Juliani et al. 2020) provide 3D environments and user interfaces to train neural networks virtually to perform specific tasks in well-documented and accessible ways today.

To leverage these new tools, fast and native links with the robot simulation/control software will be required.

6.3 Challenges for Creative Robotics

The challenges of the field of creative robotics go beyond the area of software, as most other aspects of industrial robotics are also closely aligned with the specific needs and requirements of large industries rather than small enterprises, especially concerning safety, the tender process, and knowledge transfer in general. Solving these challenges will require a coordinated effort by researchers, industry, and the greater field of robotics.

6.3.1 Safety

Robot safety is a significant challenge encountered by the creative industry, with its generally smaller budgets than industry. As an “incomplete machine” (European Parliament and Council 2006), a robotic arm only becomes a machine in the legal sense once it is equipped with the process-specific tool and then needs to be approved for operation by a qualified person (“CE”). While that process is not a requirement for research and development, it adds significant costs to the initial purchasing costs of the robot. In industry, involving a safety expert in the early planning stages is considered imperative. However, in the creative industry, this is often omitted for smaller installations,

which can lead to high costs for making existing cells conform to norms or put the owner at risk of legal liability in the event of an accident. New sensors such as certified 3D cameras are expected to facilitate the situation for the creative industry by having the user define safe spaces in a single configuration environment rather than wiring sets of magnetic and optical sensors to supervise a workspace. Ideally, it will be possible to implement the setup of the safety zones directly in the simulation environment or, in the future, even adapt safe spaces parametrically according to the current toolpath.

6.3.2 Experimentation and Acquisition

Another challenge for the creative industry is the standard tender process for robotic processes: The client needs to create a detailed specification sheet, based on which a robot integrator creates an individualized offer. The integrator acts as a general contractor, commonly purchasing all individual components (robotic arms, PLCs, sensors, actuators, etc.) and setting them up at the client's location. While creating the specifications is comparably easy for standard processes like milling and pick-and-place, it becomes extremely hard to do the specifications for a non-standard process that has not yet been developed.

To facilitate that process, companies like KUKA set up application centers where new processes can be evaluated in a controlled environment. At the same time, well-funded maker spaces, like the Grand Garage in Linz, Austria, are setting up specific robot labs where paying members can explore new processes at their own pace. Making simulation tools such as KUKA|prc widely and easily available also supports the process of properly defining the parameters of a project.

6.3.3 Knowledge Transfer

One challenge of using robots across all industries is the absence of publicly accessible documentation. In the case of KUKA, basic documentation is provided online. However, advanced documentation is only supplied in specialized courses that KUKA Colleges offer. This lack of documentation leads to a situation where knowledge is pooled

among senior robot users, and publicly accessible knowledge is primarily available through user-organized forums, where robot users can ask for assistance from their peers.

Engaging new users is, therefore, highly dependent on accessible interfaces. This can be observed in recent robot architectures like KUKA Sunrise (Chapter 4.7) that open up robotics to new groups using modern, application-agnostic programming languages like Java, allowing users to apply their programming knowledge to the machine more rapidly. In the particular case of KUKA Sunrise, that development is mainly focused on the academic area and industrial research and development due to the high cost of the platform that excludes it for many creative applications.

Within the scope of the creative industry, the formation of communities through efforts by institutions like the Association for Robots in Architecture is creating ways for new robot users to engage with more experienced users and exchange concepts, ideas, and solutions within their own disciplines, fostering the above-mentioned machine knowledge.

6.3.4 Ethical Considerations

Robotics and automation, especially concerning artificial intelligence, are increasingly starting to touch our everyday lives. Analysts at McKinsey (Bughin et al. 2018) predict that by 2030, the requirement for technological skills will increase by 55% while the demand for physical and manual skills will drop by 14 percent. However, within the scope of the creative industries, it has to be noted that creativity as a skill is assigned to the category of higher cognitive skills and is expected to rise by 14-19%. The World Economic Forum (2016) also predicts a rising demand for architects in the future job market.

This matches with recent studies that present the shortage of skilled workers as the greatest challenge for the crafts and trades (Faißt, Hamann, and Jahn 2020) rather than automation. As such, robotics may be a way for those fields to not only support their fabrication processes towards mass customization but also appear as a more attractive, innovative employer.

Frequently discussed ethical questions, such as the trolley problem concerning autonomous vehicles (Thomson 1976), do not apply to creative applications of robotics, and neither do ethical challenges of collaborative robotics, such as performance monitoring (Fletcher and Webb 2017), due to the generally small size of the companies.

The author hopes that robotics in the creative industries will lead to more local fabrication and value creation, fostering new jobs and innovation.

6.4 Vision

By design, a large part of this thesis looks back at the prior research that led to the current state of the developed software. But where does the field go from here? Where may the future of robots in the creative industry be?

A field of importance will be distributed fabrication, where one can see similarities to cloud computation. Not too many years ago, hosting a popular website or managing a large company's data required high-end hardware that was only available to users with similarly high budgets. With the rise of cloud computing, companies do not need to host their own servers; instead, they use the services of specialized companies running huge server parks that offer easy scalability and high performance.

By making these systems available to end-users, they were, in a sense, democratized. At the same time, this high demand for computational power combined with the promise of scalability has led to overcapacities, which are provided to individual users at a comparably low cost.

Applied to the field of creative robotics, there is a great potential for providing scalable fabrication (Figure 63) to individual customers so that, e.g., products may then be fabricated on demand, on multiple sites, close to the customer, paying only for the machine time used.

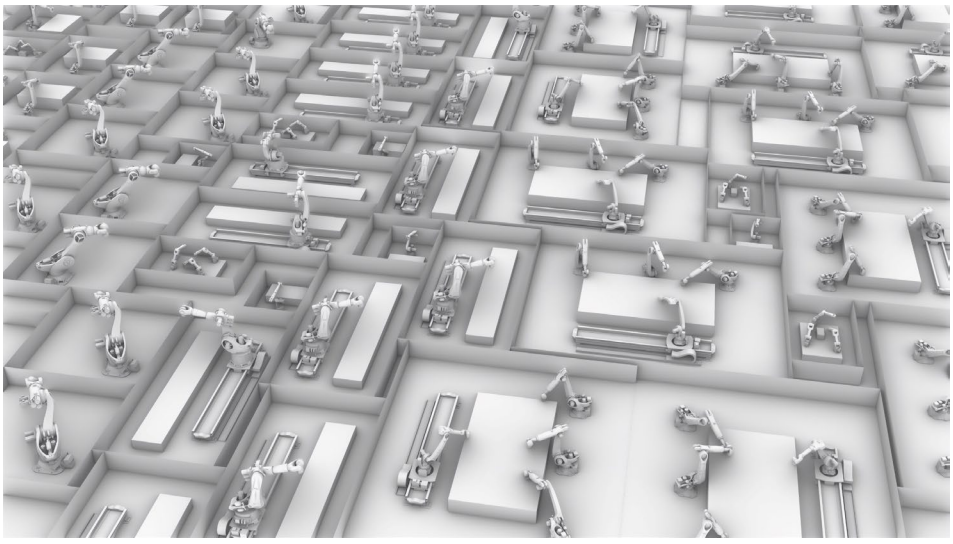

Figure 63: Conceptual image showing scalable robotic cells / microfactories.

This is already done by additive manufacturing service providers, where the enabling factor is not just the hardware but the intelligent software that can ensure the feasibility of designs and streamline the data- and workflows. The use of robots has the potential to expand significantly the range of products that may be fabricated, which is a first step towards providing assembly for 3D-printed components.

At a later stage, generic robot cells may be customized entirely at short notice to new requirements. This leads to many challenges, from relatively clearly defined tasks such as grasping arbitrary objects to extensive topics such as logistics.

A building part of such a scalable "robot park" is the micro-factory, a concept that is already being explored today, e.g., within the context of urban manufacturing, where traditional production lines take up too much space, and the use of robot arms promises more compact, multifunctional fabrication concepts.

To realize creative micro-factories, it will be necessary to develop entirely new dataflows that combine creative users' customization from tools like Grasshopper with the high reliability of control systems used for industrial mass fabrication. Off-site fabrication will require new ways of managing machines remotely – going beyond the current approaches in an industry where engineers may log into remote systems

for troubleshooting towards enabling the remote commissioning of entirely new production processes.

A significant challenge will be mobility beyond factories and out in the field. A robot moving around is not complicated by itself; the challenge lies in understanding its environment (Figure 64). To work in an environment like a construction site, a robot must rely on real-time data rather than on idealized 3D plans while at the same time "understanding" the environment, as only the environment provides the context a machine needs to execute commands.

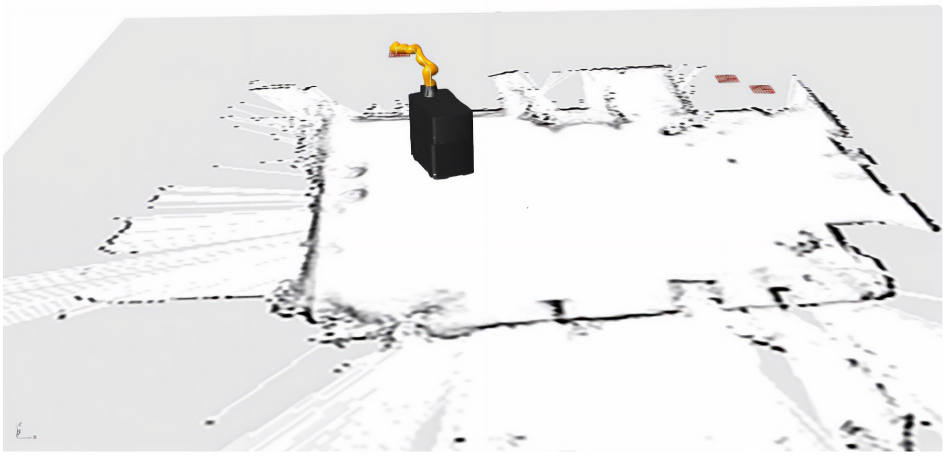

Figure 64: Perception of a lab environment at RWTH Aachen by an KMR iiwa platform in KUKA|prc.

Research has already solved many problems in that field. However, the question remains how that knowledge may be made available to users who do not specialize in computer vision. While architects may already define construction tasks within a BIM environment, the robot would then have to intelligently fit the instructions to the actual environment of the construction site, allowing for construction tolerances and other imperfections that are not reflected in the planning.

Artificial intelligence and machine learning are essential technologies to achieve efficient recognition, segmentation, and interpretation of complex environments. It is necessary to move beyond CAD-data-based object recognition, such as that offered by PhotoNeo, and develop

accessible vision products that can work on a larger scale in varied and unstructured environments.

Understanding its environment along with more complex voice and gestural recognition will also be a key to robotic assistants who may at some point support a carpenter in building furniture or moving heavy objects for a mason.

Finally, with the maturing of the area of creative robotics, there will be more task-specific machines. By design, robotic arms are universal machines and, therefore, never as efficient as a machine built specifically for a single operation. For a long time, creating new machines required high-end electrical and mechanical engineering knowledge until physical computing platforms like Arduino rapidly opened up the broader field of robotics to new users. A critical development is the increasing availability of powerful modules, rather than individual electrical components, that can be combined into larger, reliable systems.

What ties all aspects mentioned in this section together is that they are already technically possible, just limited in scale or application. But once researchers manage to get them scaled up and exposed through accessible software interfaces, they have the potential to rapidly transform entire creative fields and make ideas that initially seemed far-fetched possible from one day to another.

Just as they do now, robots will then provide new ways for that new digital data to affect our physical environment.

Appendix

Glossary

Agilus
Series of compact (3-10kg payload) robots by KUKA.

BIM
Building Information Modeling.

C#
General-purpose programming language initially developed by Microsoft.

CAD
Computer Aided Design.

CAM
Computer Aided Manufacturing.

Cobot
Collaborative robot, an industrial robot with special sensors to facilitate a safe collaboration with human workers.

CNC
Computer Numeric Control.

Dynamo
Visual, flow-based programming for Autodesk Revit.

Ethernet.KRL
Software developed by KUKA to allow TCP and UDP communication with a KUKA robot.

G-Code
Widely used CNC programming language.

Generative Components
Visual, flow-based programming in Bentley Microstation.

Grasshopper
Visual, flow-based programming in McNeel Rhinoceros.

HoloLens
Mixed reality headset by Microsoft.

iiwa
Intelligent industrial work assistant. Series of collaborative robots by KUKA.

IO
Inputs/Output.

KRL
KUKA Robot Language, proprietary programming language used to program KUKA robots.

KSS
KUKA System Software, the operating system software used for industrial robots by KUKA.

KUKA
Germany-based manufacturer of industrial robots.

KUKA|prc
Software for parametric robot control of KUKA robots.

LIN
Robot movement using linear interpolation.

mxAutomation
Software developed by KUKA to allow PLCs and PCs to control KUKA robots remotely.

.NET
Cross-platform developer platform originally created by Microsoft.

PLC
Programmable Logic Controller. An industrial digital computer.

PTP
Robot movement using point-to-point interpolation

Quantec
Series of high-payload (120-310kg) industrial robots by KUKA.

Rhinoceros 3D
CAD software developed by McNeel.

RhinoCommon
Geometry library used by Rhinoceros 3D.

Robots in Architecture
Non-profit association founded by Sigrid Brell-Cokcan and Johannes Braumann.

RSI
Robot Sensor Interface. Software developed by KUKA for real-time control of a KUKA robot.

Servoing
Soft real-time interface for robot control and sensor integration of a KUKA cobot.

smartPad
Teach pendant with touchscreen for interacting with the industrial robot.

SPL
Robot movement using spline interpolation.

Sunrise
Operating system software used for collaborative robots by KUKA.

TCP
Transmission Control Protocol. Communications protocol for reliable, ordered, and error-checked byte streams

Tool Center Point.

UDP
User Datagram Protocol. Communications protocol for low-latency, loss-tolerant connections.

Unity
Cross-platform game engine.

XML
Extensible Markup Language.

Exemplary KUKA|prc Workflows in Grasshopper

The following presents two exemplary workflows that demonstrate a typical user interaction. The first example focuses on the geometric aspects of robot programming, while the second focuses on multi-stage processes.

Hotwire Cutting: Hotwire cutting is a relevant process for architecture and design, as it allows the users to cut through a light but stable foam like EPS or XPS in a very controlled and fast way without producing large amounts of waste or noise. It is, therefore, a ubiquitous tool in architecture schools for model building. Still, on a larger scale, it is also interesting for full-scale fabrication, as exhibited in the work of Odico (McGee, Feringa, and Søndergaard 2013), a startup specifically founded to develop large-scale, robotic hotwire cutting processes as well as the author's research (Brell-Cokcan and Braumann 2014). On an industrial scale, hotwire cutting is currently mainly used to customize the size of foam blocks, seldomly creating non-planar shapes. The geometric challenge of working with a hotwire cutter is that a straight line can represent the idealized hot wire. As such, it is only possible to create ruled surfaces, i.e., surfaces that can be swept out by moving a line in space. The literature does not show any large-scale applications of non-linear hotwire cutting tools.

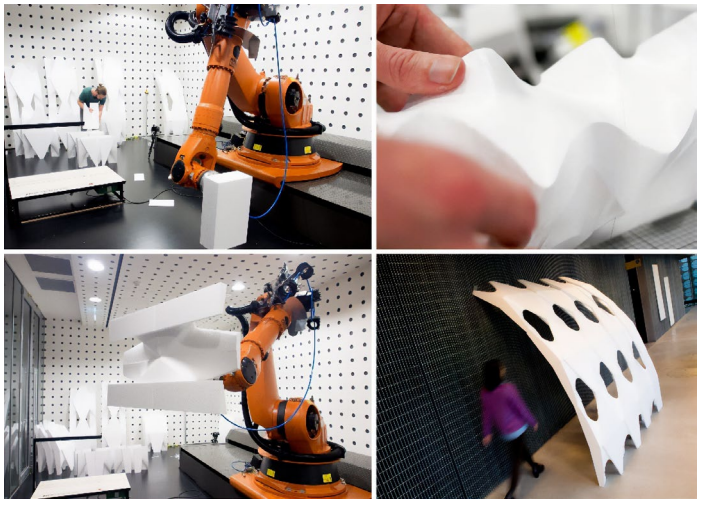

Figure 65: Hotwire-cutting as part of a workshop at RMIT, Australia, 2014.

In this example, the surface that should be hotwire-cut is generated as a loft between two NURBS curves, resulting in a ruled surface. The goal is to create toolpaths whose Z-axis follows the ruling of the surface. To do so, the tool needs to be defined, whose geometry is drawn as a mesh and placed at the global origin of the CAD model, which for a tool geometry represents the robot flange coordinate system. The reference point of the tool is measured in the middle of the hotwire cutter, resulting in X and Y being 0 and Z representing the distance between the TCP and the flange. As KUKA|prc uses X as the tool axis, the hotwire would now be perpendicular to the rulings of the surface. Therefore, the tool axis needs to be adjusted with a rotation, adding either positive or negative 90 degrees to the A value, i.e., the rotation around the normal of the robot flange.

The calibration of the base coordinate system takes place at the physical robot by measuring a corner of the foam block, a point in X, and a point in Y direction with a measuring tip. The robot controller calculates a coordinate system through that data, expressed in XYZABC values, which the user can enter in the KUKA|prc Settings. The robot is now transformed inversely so that the foam block is at the origin of the Rhinoceros coordinate system, and the robot is offset accordingly.

To generate the toolpaths, the rulings must be extracted from the ruled surface. These rulings are equivalent to the isocurves of NURBS surfaces (though isocurves may be free-formed, while rulings are straight lines); as such, it is possible to use the *Iso Curve* component and connect the surface to the Surface input. The specific location of the isocurve is defined via the UV point input, which describes a position within the surface's local UV coordinate system, where U can be considered a local X value and V a local Y value. Therefore, UV values can be expressed as a regular 3D vector with XYZ parameters. As multiple rulings need to be extracted, it is not sufficient to provide a single coordinate; instead, the *Range* component generates a list of values whose domain and number of values can be set parametrically, e.g., via sliders. To facilitate the process, rather than extracting the parametrization of the surface and then adjusting the UV domains accordingly, the option to reparametrize the NURBS surface to a range of 0.0-1.0 is activated. This generates a list of line geometries from which a toolpath can be created.

Every robot position needs to be defined as a coordinate system or *Plane*, as the geometry type is referred to in Rhinoceros 3D. The plane's origin can be extracted by connecting the lines to an *Evaluate Length* component. This component extracts a position on a curve, either using absolute or normalized values. The middle point of the curves is extracted by setting the component to using normalized units with 0.5 as the input. Therefore, the input list of lines generates an equal number of middle points.

Grasshopper provides a *Plane Normal* component, which constructs a plane geometry from a point and a normal vector. The middle points are set as the point input. The *Vector 2Pt* component is used to define the normal vector, which calculates a vector by subtracting one point coordinate from the other. These points are extracted by plugging the lines into the *End Points* component, which outputs the start and endpoint of a curve geometry.

Geometrically, defining a point and a normal vector is not sufficient to unambiguously construct a coordinate system; as such, one can observe that the X and Y axes are not constrained and switch at inflection points. The KUKA|prc *Orient Plane* component rotates the coordinate system around its normal vector so that the X-axis faces away from a given position connected to the Orientation Point input while keeping the origin and normal of a coordinate system intact.

To transform the planes into robot commands, they are connected to the Plane input of a *LINear Movement* component so that the robot interpolates in a straight line between positions. Once the movement data is plugged into the *Core* component, a number slider, generating values between 0 and 1, can be used to go through the simulation.

The simulation shows immediately that during the approach, the hotwire cutter cuts through the block of material, which is not a desired behavior. A *Cartesian Offset* component is added, which works on a list of movements and duplicates the first and last movement, transforming it by XYZ according to the input data. In this case, with a Z offset of 150mm, the hotwire starts 150mm above the first position, moves downwards, cuts, and then moves upwards by another 150mm before going to its end position with a PTP movement.

The *Core* component is drawn in red if the toolpath contains unreachable positions or collisions. Yellow signifies possible singularities, and grey has no immediate problems. One can, therefore, interact intuitively with the parametric model until the component is displayed in red. To enable a more structured approach to problem-solving, the Analysis tab of the settings interface shows a plot where the axis movement is plotted against time, with red and yellow bars precisely showing unreachable positions or positions close to singularities.

In this specific use case, given a desired geometry, two particular degrees of freedom can be used to optimize the reachability of the cutting process: The position of the block of material in relation to the robot (or inversely) – within the physical constraints of our workspace – and the position of the orientation point, seeing that the idealized hotwire can be considered a rotational symmetric tool. In this case, a decent orientation would be above the side of the block closer to the robot. Therefore, the tool would be vertical when close to the robot – avoid collisions with the machine – and tilt towards the robot the further its position is away from the point, improving reachability.

If the position of the workpiece, and by that the XYZABC coordinates of the base coordinate system, are changed, it is imperative to ensure that the XYZABC values on the robot and in the simulation match, as otherwise, it is impossible to get an accurate, virtual simulation of the process.

By default, once a project name and an output file directory have been set, KUKA|prc writes KRL code as SRC files in the background whenever code changes occur. For immediate access to the robot, the files can be saved on an SMB share on the robot and then only need to be copied from the hard drive to the robot's RAM drive.

Pick-and-Place: Pick-and-Place applications belong to the most frequently seen robot applications in industry but are commonly programmed once, e.g., for placing objects into trays, and then never changed. More flexible programming strategies now allow the user to define such a process more dynamically. This example stacks a sequence of bricks pioneered by ETH Zurich in 2006 (Bonwetsch et al.

2006). It went far beyond the state of the art at the time but can now be achieved by non-expert users.

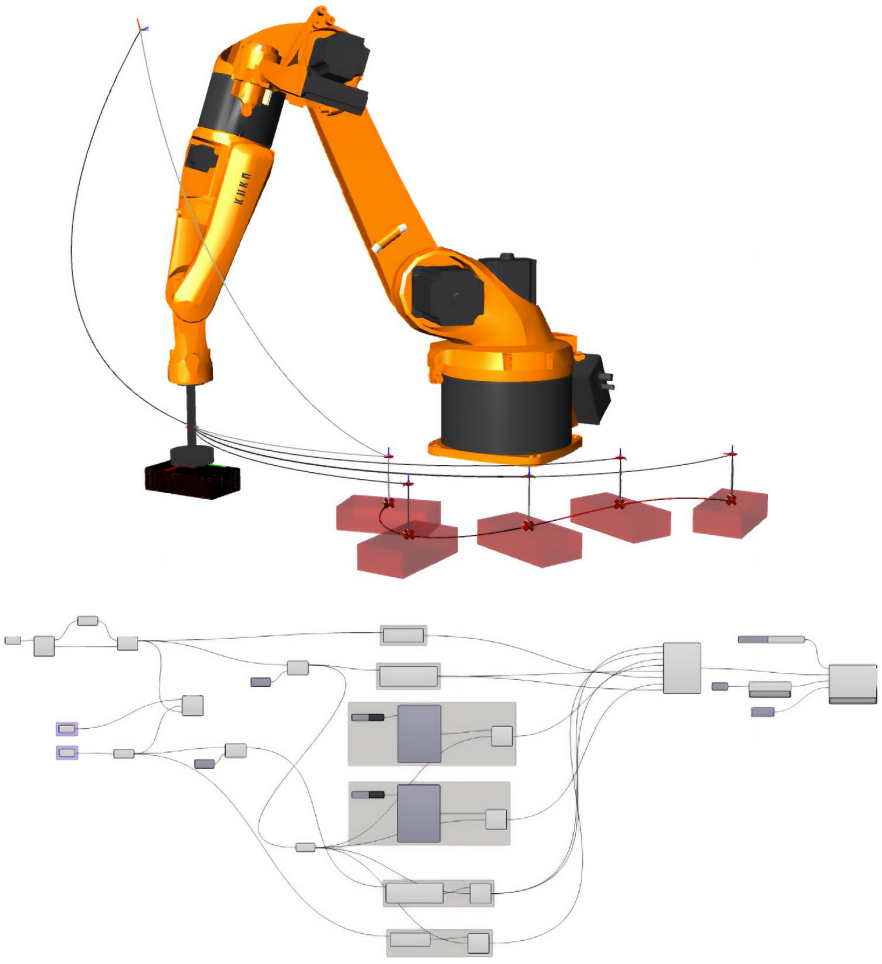

Figure 66: Pick-and-place cycle programmed using Grasshopper and KUKA|prc.

The starting geometry is a given gripper tool geometry with known XYZABC calibration values, a plane signifying the position where bricks are fed to the robot, and a mesh model of the brick. In this example, the bricks are aligned on a curve instead of a complex brick-bond algorithm. To do so, a NURBS curve is drawn in Rhino and referenced in Grasshopper. The *Divide Curve* component is used with a numeric value to generate a list of points, each of which a coordinate system is placed

using the *Plane XY* component. In addition to the point coordinates, the component outputs the tangent vector as unitized XYZ values for each position. This makes it possible to align our planes with the vector through the *Align Plane* component, generating the target position for our bricks. With all geometric data in place, the order of operations for a pick-and-place operation must be defined. Significant are safety offsets to ensure that bricks are placed flat and without collisions. While a PTP movement will arrive at the same position as a LIN movement, the PTP movement will most likely tilt the brick out of the XY plane, causing it to settle unevenly or even causing a collision event. Thus, the following structure is used:

- PTP movement to position above the brick station.
- LIN movement to pick-up point.
- Send command to close the gripper.
- Wait until the gripper has closed.
- LIN movement to pick-up point.
- PTP movement to position above target position.
- LIN movement to target position.
- Send command to open gripper.
- Wait until the gripper has opened.
- LIN movement to position above target position.

The wait commands are necessary because a physical gripper always takes at least a few hundred milliseconds to close. It also ensures that the programmed position is reached and no motion blending occurs, which would otherwise cause the robot to reverse its path before reaching the programmed position.

In a text-based programming system, this logic could be placed inside a for-loop that iterates over the list of targets, creating a pick-and-place process for each target position. For a process in visual programming, the data structure has to be changed to achieve the same result. The *Command Weaver* component takes the same number of KUKA|prc

commands for each input and weaves them into one output, putting the first command from the first input first, followed by the first command of the second input and the first command of the third input, etc. To do so, the number of target positions has to be defined – in this case, the number input of the *Divide Curve* component plus 1 (assuming an open curve) – so that all single commands may be duplicated by the number with the *Duplicate Data* component. This results in a data structure that the *Command Weaver* can transform into the correct, single list of commands to perform the process.

Compared to the previous process, the current strategy has less potential for optimization since the tool is not rotationally symmetrical. Assuming that the brick is gripped centrally, it would be possible to rotate each brick by 180 degrees without changing the visual result of the process. However, as axis 6 is usually the fastest axis of a robotic arm, even a significant rotation by 180 degrees might not significantly impact the general cycle time, as the other axes will most likely still be considerably slower. Reorienting bricks around their Z-axis might minimize the risk of winding up the gripper's various cables and hoses, whose behavior is not simulated within KUKA|prc.

References

ABB. 2020a. *Machining PowerPac*.

———. 2020b. *RobotStudio*. https://new.abb.com/products/robotics/robotstudio.

Aish, Robert, and Robert Woodbury. 2005. “Multi-Level Interaction in Parametric Design.” In , 3638:151–62. https://doi.org/10.1007/11536482_13.

Attig, Christiane, Nadine Rauh, Thomas Franke, and Josef F. Krems. 2017. “System Latency Guidelines Then and Now – Is Zero Latency Really Considered Necessary?” In *Engineering Psychology and Cognitive Ergonomics: Cognition and Design*, edited by Don Harris, 3–14. Lecture Notes in Computer Science. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-58475-1_1.

Ault, Holly K, and Arnold Phillips. 2016. “Direct Modeling: Easy Changes in CAD?” In *ASEE EDGD 70th Midyear Conference*. Daytona Beach Shores.

Autodesk. 2020a. *3ds Max*. https://www.autodesk.com/products/3ds-max/overview.

———. 2020b. *AutoCAD*. https://www.autodesk.com/products/autocad/overview.

———. 2020c. *PowerMill Robot*. https://manufacturing.autodesk.com/solutions/robotics/.

Bedi, Sanjeev, Stephen Mann, and Cornelia Menzel. 2003. “Flank Milling with Flat End Milling Cutters.” *Computer-Aided Design* 35: 293–300.

Bentley Systems. 2003. *Generative Components*.

Bock, Thomas. 1988. “Innovationen Im Bauwesen: Roboter Auf Japanischen Baustellen.” *Innovationen Im Bauwesen: Roboter Auf Japanischen Baustellen* 63 (3): 121–24.

———. 2008. “Construction Automation and Robotics.” In . https://doi.org/10.5772/5861.

Bonwetsch, Tobias, Fabio Gramazio, and Matthias Kohler. 2007. “Digitally Fabricating Non-Standardised Brick Walls.” In *ManuBuild, 1st International Conference*, 191–96. Rotterdam.

Bonwetsch, Tobias, Daniel Kobel, Fabio Gramazio, and Matthias Kohler. 2006. “The Informed Wall: Applying Additive Digital Fabrication Techniques on Architecture.” In *Synthetic Landscapes*, 489–95. Louisville.

Bradski, G. 2000. “The OpenCV Library.” *Dr. Dobb’s Journal of Software Tools*.

Brandt, Eli, and Roger B Dannenberg. 1998. “Low-Latency Music Software Using off-the-Shelf Operating Systems.”

Braumann, Johannes. 2020. “Robots for Skill Digitisation.” In *Design Transactions: Rethinking Information for a New Material Age*, edited by R. Sheil, M. Thomsen, M. Tamke, and S. Hanna. London, UK: UCL Press. https://doi.org/10.14324/111.9781787355026.

Braumann, Johannes, and Sigrid Brell-Cokcan. 2011. “Parametric Robot Control: Integrated CAD/CAM for Architectural Design.” In *Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture*, 242–51. Banff.

———. 2012. “Digital and Physical Tools for Industrial Robots in Architecture: Robotic Interaction and Interfaces.” *International Journal of Architectural Computing Vol. 10 - No. 4, 541-554*. http://papers.cumincad.org/cgi-bin/works/paper/ijac201210405.

———. 2014. “Visual Robot Programming - Linking Design, Simulation, and Fabrication.” *Simulation Series* 46 (January): 118–25.

———. 2015. “Adaptive Robot Control - New Parametric Workflows Directly from Design to KUKA Robots.” In *Real Time - Proceedings of the 33rd ECAADe Conference*, 243–50. CUMINCAD.

———. 2018. “Accessible Robotics: Enabling Industry and the Creative Community.” In *Towards a Robotic Architecture*, edited by Mahesh Daas and Andrew John Wit. Applied Research and Design Publishing.

Braumann, Johannes, Sven Stumm, and Sigrid Brell-Cokcan. 2016. “Towards New Robotic Design Tools: Using Collaborative Robots within the Creative Industry.” In *Proceedings of the 36th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, 164–73. Ann Arbor.

Brell-Cokcan, Sigrid. 2017. Kunsthaus Graz-Up into the Unknown: Peter Cook, Colin Fournier and the Kunsthaus.

Brell-Cokcan, Sigrid, and Johannes Braumann. 2010. “A New Parametric Design Tool for Robot Milling.” In *LIFE in:Formation, On Responsive Information and Variations in Architecture*, 357–63.

———. 2013a. “Industrial Robots for Design Education: Robots as Open Interfaces beyond Fabrication.” In *Global Design and Local Materialization*, edited by Jianlong Zhang and Chengyu Sun, 109–17. Communications in Computer and Information Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-642-38974-0_10.

———, eds. 2013b. *Rob|Arch 2012: Robotic Fabrication in Architecture, Art and Design*. Wien: Springer-Verlag. https://doi.org/10.1007/978-3-7091-1465-0.

———. 2014. “Robotic Production Immanent Design: Creative Toolpath Design in Micro and Macro Scale.” In *Proceedings of the 34th Annual Conference of the Association for Computer Aided Design in Architecture (ACADIA)*, 10.

———. 2015. “Toward Adaptive Robot Control Strategies.” In *ACADIA 2105: Computational Ecologies: Design in the Anthropocene*, 223–31. Cincinnatti.

———. 2018. “Making Robots Accessible.” In *Proceedings of the 33rd Annual Conference of the Association for Computer Aided Design in Architecture*. Mexico City.

Brell-Cokcan, Sigrid, Martin Reis, Heinz Schmiedhofer, and Johannes Braumann. 2009. “Digital Design to Digital Production.” In *Computation: The New Realm of Architectural Design*, 8. Istanbul.

Bughin, James, Eric Hazan, Susan Lund, Peter Dahlström, Anna Wiesinger, and Amresh Subramaniam. 2018. “Skill Shift: Automation and the Future of the Workforce.” McKinsey Global Institute.

Bull, Stuart, and Steve Downing. 2004. “Beijing Water Cube - The IT Challenge.” *The Structural Engineer* 82 (13). https://www.researchgate.net/publication/292689518_Beijing_Water_Cube_-_The_IT_challenge.

C3D Labs. 2020. *C3D Toolkit*. https://c3dlabs.com/en/products/c3d-toolkit/.

Caldera, Shehan, Alexander Rassau, and Douglas Chai. 2018. “Review of Deep Learning Methods in Robotic Grasp Detection.” *Multimodal Technologies and Interaction* 2 (3): 57. https://doi.org/10.3390/mti2030057.

Carvalho, Thyago P., Fabrízzio A. A. M. N. Soares, Roberto Vita, Roberto da P. Francisco, João P. Basto, and Symone G. S. Alcalá. 2019. “A Systematic Literature Review of Machine Learning Methods Applied to Predictive Maintenance.” *Computers & Industrial Engineering* 137 (November): 106024. https://doi.org/10.1016/j.cie.2019.106024.

Catmull, Edwin, and Raphael Rom. 1974. “A Class of Local Interpolating Splines.” In *Computer Aided Geometric Design*, edited by ROBERT E. Barnhill and RICHARD F. Riesenfeld, 317–26. Academic Press. https://doi.org/10.1016/B978-0-12-079050-0.50020-5.

Celani, Gabriela, and Carlos Eduardo Verzola Vaz. 2012. “CAD Scripting and Visual Programming Languages for Implementing Computational Design Concepts: A Comparison from a Pedagogical Point of View.” *International Journal of Architectural Computing* 10 (1): 121–37. https://doi.org/10.1260/1478-0771.10.1.121.

Chang, Liang. 2015. “The Software Behind Frank Gehry’s Geometrically Complex Architecture.” Priceonomics. 2015. http://priceonomics.com/the-software-behind-frank-gehrys-geometrically/.

Chen, Heping, Weihua Sheng, Ning Xi, Mumin Song, and Yifan Chen. 2002. “CAD-based Automated Robot Trajectory Planning for Spray Painting of Free-form Surfaces.” *Industrial Robot: An International Journal* 29 (5): 426–33. https://doi.org/10.1108/01439910210440237.

Chen, I-Ming, Raymond Tay, Shusong Xing, and Song Huat Yeo. 2004. “Marionette: From Traditional Manipulation to Robotic Manipulation.” In *International Symposium on History of Machines and Mechanisms*, 119–33. Springer.

Conway, Melvin E. 1963. “Design of a Separable Transition-Diagram Compiler.” *Communications of the ACM* 6 (7): 396–408. https://doi.org/10.1145/366663.366704.

Cycling ’74. 2020. *Max*. https://cycling74.com/products/max/.

Da Silveira, Giovani, Denis Borenstein, and Flávio S Fogliatto. 2001. “Mass Customization: Literature Review and Research Directions.” *International Journal of Production Economics* 72 (1): 1–13. https://doi.org/10.1016/S0925-5273(00)00079-7.

Dassault. 2020. *CGM Core Modeler*. https://www.spatial.com/products/cgm.

Dassault Systèmes. 2020. *CATIA*. https://www.3ds.com/products-services/catia/.

Davis, Daniel, Jane Burry, and Mark C. Burry. 2011. “Understanding Visual Scripts: Improving Collaboration through Modular Programming.” https://doi.org/10.1260/1478-0771.9.4.361.

Day, Martyn. 2004. “Review: Digital Project from Gehry Technologies.” 2004. http://www.aecnews.com/articles/842.aspx.

Denavit, J., and R. S. Hartenberg. 1955. “A Kinematic Notation for Lower-Pair Mechanisms Based on Matrices.” *Trans.\ ASME E, Journal of Applied Mechanics* 22 (June): 215–21.

Devol, Jr George C. 1961. Programmed article transfer. United States US2988237A, filed December 10, 1954, and issued June 13, 1961.

Diankov, Rosen. 2010. “Automated Construction of Robotic Manipulation Programs.” PhD Thesis, Carnegie Mellon University, Robotics Institute.

DIN. 1988. “DIN 66025-2:1988-09, Industrielle Automation; Programmaufbau Für Numerisch Gesteuerte Arbeitsmaschinen; Wegbedingungen Und Zusatzfunktionen.” Beuth Verlag GmbH. https://doi.org/10.31030/2270064.

Eastman, Charles. 1974. “An Outline of the Building Description System. Research Report No. 50,” September.

Eastman, Chuck, Paul Teicholz, Rafael Sacks, and Kathleen Liston. 2008. *BIM Handbook: A Guide to Building Information Modeling for Owners, Managers, Designers, Engineers and Contractors*. Wiley Publishing.

Elkady, Ayssam, and Tarek Sobh. 2012. “Robotics Middleware: A Comprehensive Literature Survey and Attribute-Based Bibliography.” Review Article. Journal of Robotics. Hindawi. 2012. https://doi.org/10.1155/2012/959013.

Epic Games. 2020. *Blueprints Visual Scripting*. https://www.unrealengine.com/.

Ericson, Christer. 2004. *Real-Time Collision Detection*. HAR/CDR edition. Amsterdam ; Boston: CRC Press.

European Parliament and Council. 2006. “Directive 2006/42/EC on Machinery.”

Faißt, Christian, Silke Hamann, and Daniel Jahn. 2020. “Die Bedeutung des Handwerks in Baden- Württemberg – Fokus: Fachkräfte in Handwerksberufen.” Institut für Arbeitsmarkt- und Berufsforschung.

Feldman, J. A., G. M. Feldman, G. Falk, G. Grape, J. Pearlman, I. Sobel, and J. M. Tenebaum. 1969. “The Stanford Hand-Eye Project.” In *Proceedings of the 1st International Joint Conference on Artificial Intelligence*, 521–26. IJCAI’69. Washington, DC: Morgan Kaufmann Publishers Inc.

Fletcher, S. R., and P. Webb. 2017. “Industrial Robot Ethics: The Challenges of Closer Human Collaboration in Future Manufacturing Systems.” In *A World with Robots: International Conference on Robot Ethics: ICRE 2015*, edited by Maria Isabel Aldinhas Ferreira, Joao Silva Sequeira, Mohammad Osman Tokhi, Endre E. Kadar, and Gurvinder Singh Virk, 159–69. Intelligent Systems, Control and Automation: Science and Engineering. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-46667-5_12.

Fornes, Marc. 2006. “Theverymany.” 2006. http://www.theverymany.net/.

Fryling, Meg. 2019. “Low Code App Development.” *Journal of Computing Sciences in Colleges* 34 (6): 119.

Garofalo, Raffaele. 2011. *Building Enterprise Applications with Windows Presentation Foundation and the Model View ViewModel Pattern*. Microsoft Press.

Glaeser, Georg, and Franz Gruber. 2007. “Developable Surfaces in Contemporary Architecture.” *Journal of Mathematics and the Arts* 1 (1): 59–71. https://doi.org/10.1080/17513470701230004.

Glaeser, Georg, and Hellmuth Stachel. 1999. *Open Geometry: OpenGL® + Advanced Geometry*. New York: Springer-Verlag. https://doi.org/10.1007/978-1-4612-1428-1.

Glymph, James, Dennis Shelden, Cristiano Ceccato, Judith Mussel, and Hans Schober. 2004. “A Parametric Strategy for Free-Form Glass

Structures Using Quadrilateral Planar Facets." *Automation in Construction*, Conference of the Association for Computer Aided Design in Architecture, 13 (2): 187–202. https://doi.org/10.1016/j.autcon.2003.09.008.

Gu, Yan, Yong He, Kayvon Fatahalian, and Guy Blelloch. 2013. "Efficient BVH Construction via Approximate Agglomerative Clustering." In *Proceedings of the 5th High-Performance Graphics Conference*, 81–88. HPG '13. Anaheim, California: Association for Computing Machinery. https://doi.org/10.1145/2492045.2492054.

Harris, Alan. 2010. "Scripting via IronPython." In *Pro ASP.NET 4 CMS: Advanced Techniques for C# Developers Using the .NET 4 Framework*, edited by Alan Harris, 197–228. Berkeley, CA: Apress. https://doi.org/10.1007/978-1-4302-2713-7_7.

Helbig, Tobias, Stefan Erler, Engelbert Westkämper, and Johannes Hoos. 2016. "Modelling Dependencies to Improve the Cross-Domain Collaboration in the Engineering Process of Special Purpose Machinery." *Procedia CIRP* 41: 393–98. https://doi.org/10.1016/j.procir.2015.12.123.

Hietanen, Antti, Alireza Changizi, Minna Lanz, Joni Kamarainen, Pallab Ganguly, Roel Pieters, and Jyrki Latokartano. 2019. "Proof of Concept of a Projection-Based Safety System for Human-Robot Collaborative Engine Assembly." In *2019 28th IEEE International Conference on Robot and Human Interactive Communication (RO-MAN)*, 1–7. New Delhi, India: IEEE. https://doi.org/10.1109/RO-MAN46459.2019.8956446.

Hua, Hao. 2019. *JavaKUKA*. https://github.com/whitegreen/JavaKUKA.

Hypertherm. 2020. *Robotmaster*. https://www.robotmaster.com/.

IFR. 2018. "World Robotics 2018 - Industrial Robots."

Iglesias, I., M.A. Sebastián, and J.E. Ares. 2015. "Overview of the State of Robotic Machining: Current Situation and Future Potential." *Procedia Engineering* 132: 911–17. https://doi.org/10.1016/j.proeng.2015.12.577.

Iqbal, M, and M. S. J Hashmi. 2001. “Design and Analysis of a Virtual Factory Layout.” *Journal of Materials Processing Technology* 118 (1): 403–10. https://doi.org/10.1016/S0924-0136(01)00908-6.

ISO. 1998. *ISO 9283:1998*. https://www.iso.org/cms/render/live/en/sites/isoorg/contents/data/standard/02/22/22244.html.

Jalba, C K, P Konold, I Rapp, C Mann, and A Muminovic. 2017. “Cooperation between Humans and Robots in Fine Assembly.” *IOP Conference Series: Materials Science and Engineering* 163 (January): 012049. https://doi.org/10.1088/1757-899X/163/1/012049.

Juetten, Mary. 2018. “Failed Startups: Rethink Robotics.” Forbes. 2018. https://www.forbes.com/sites/maryjuetten/2018/12/06/failed-startups-rethink-robotics/.

Juliani, Arthur, Vincent-Pierre Berges, Ervin Teng, Andrew Cohen, Jonathan Harper, Chris Elion, Chris Goy, et al. 2020. “Unity: A General Platform for Intelligent Agents.” *ArXiv:1809.02627 [Cs, Stat]*, May. http://arxiv.org/abs/1809.02627.

Kaaresoja, Topi, Stephen Brewster, and Vuokko Lantz. 2014. “Towards the Temporally Perfect Virtual Button: Touch-Feedback Simultaneity and Perceived Quality in Mobile Touchscreen Press Interactions.” *ACM Transactions on Applied Perception* 11 (July): 1–25. https://doi.org/10.1145/2611387.

Keough, Ian. 2020. *Dynamo BIM*. https://dynamobim.org/.

Kuang-Hua, Chang. 2014. *Product Design Modeling Using CAD/CAE*. Elsevier. https://doi.org/10.1016/C2012-0-00834-4.

Kucuk, Serdar, and Zafer Bingul. 2010. “An Off-Line Robot Simulation Toolbox.” *Computer Applications in Engineering Education* 18 (1): 41–52. https://doi.org/10.1002/cae.20236.

KUKA. 2004. *KUKA.CAMRob Release 1.0*.

———. 2009. *KUKA.CAMRob*.

———. 2014. “Basic Principles of Motion Programming.” In *KUKA System Software 8.3 - Operating and Programming Instructions for System Integrators*, 301–4.

———. 2015. *KUKA Sunrise.SmartServo 1.9, Sunrise.DirectServo 1.9 V1.1*.

———. 2017. *CODESYS Library for KUKA.PLC MxAutomation 3.0*.

———. 2019. *KR QUANTEC-2 Specification (Spez KR QUANTEC-2 V2)*.

———. 2020a. *KUKA.Sim*.

———. 2020b. *Ready2_pilot*. https://www.kuka.com/en-at/products/robotics-systems/ready2_solutions/kuka--ready2_pilot.

Lee, C., and M. Ziegler. 1984. “Geometric Approach in Solving Inverse Kinematics of PUMA Robots.” *IEEE Transactions on Aerospace and Electronic Systems* AES-20 (6): 695–706. https://doi.org/10.1109/TAES.1984.310452.

Lee, Joo-sung, Nahyun Kwon, Nam-hyuk Ham, Jae-jun Kim, and Yong-han Ahn. 2019. “BIM-Based Digital Fabrication Process for a Free-Form Building Project in South Korea.” *Advances in Civil Engineering* 2019 (May): 1–18. https://doi.org/10.1155/2019/4163625.

Leibowitz, Matthew. 2020. *Jitter Physics*. https://github.com/mattleibow/jitterphysics.

Lucke, Jacques. 2020. *Blender Animation Nodes*. https://github.com/JacquesLucke/animation_nodes.

Ludiq. 2020. *Bolt: Visual Scripting for Unity*. https://ludiq.io/bolt.

Martin, Patrick, Elliot Johnson, Todd Murphey, and Magnus Egerstedt. 2011. “Constructing and Implementing Motion Programs for Robotic Marionettes.” *IEEE Transactions on Automatic Control* 56 (4): 902–7.

Maslar, M. 1996. “PLC Standard Programming Languages: IEC 1131-3.” In *Conference Record of 1996 Annual Pulp and Paper Industry*

Technical Conference, 26–31. https://doi.org/10.1109/PAPCON.1996.535979.

Mateo, Carlos, Alberto Brunete, Ernesto Gambao, and Miguel Hernando. 2014. “Hammer: An Android Based Application for End-User Industrial Robot Programming.” In . https://doi.org/10.1109/MESA.2014.6935597.

Math.NET. 2020. https://github.com/mathnet.

Maxon. 2020a. *Cinema 4D*. https://www.maxon.net/en/.

———. 2020b. *XPresso*. https://www.maxon.net/en/.

Mayer, Hermann, Istvan Nagy, and Alois Knoll. 2004. “Inverse Kinematics of a Manipulator for Minimally Invasive Surgery.” *Technische Universität München*.

McGee, Wes, Jelle Feringa, and Asbjørn Søndergaard. 2013. “Processes for an Architecture of Volume.” In *Rob | Arch 2012*, edited by Sigrid Brell-Çokcan and Johannes Braumann, 62–71. Vienna: Springer. https://doi.org/10.1007/978-3-7091-1465-0_5.

McNeel. 2020a. *Rhinoceros 3D*. https://www.rhino3d.com/.

———. 2020b. “The History of Rhino [McNeel Wiki].” 2020. https://wiki.mcneel.com/rhino/rhinohistory.

Microsoft. 2020. .“.NET Standard.” 2020. https://docs.microsoft.com/en-us/dotnet/standard/net-standard.

Mikado. 2014. “Elefantenhaus Zürich,” 2014.

Mitsi, S., K.-D Bouzakis, G. Mansour, Dimitrios Sagris, and Georgios Maliaris. 2005. “Off-Line Programming of an Industrial Robot for Manufacturing.” *The International Journal of Advanced Manufacturing Technology* 26 (August): 262–67. https://doi.org/10.1007/s00170-003-1728-5.

Molfino, Rezia, Enrico Carca, Matteo Zoppi, Fabio Bonsignorio, Massimo Callegari, Andrea Gabrielli, and Marco Principi. 2008. “A Multi-Agent 3D Simulation Environment for Clothing Industry.” In *Simulation, Modeling, and Programming for Autonomous Robots*, edited by Stefano Carpin, Itsuki Noda, Enrico Pagello, Monica Reggiani, and Oskar von Stryk, 53–64. Lecture Notes in Computer Science. Berlin, Heidelberg: Springer. https://doi.org/10.1007/978-3-540-89076-8_9.

Morrison, J. Paul. 2010. *Flow-Based Programming: A New Approach to Application Development*. J.P. Morrison Enterprises.

Mower, Sarah. 2018. “Remembering the Potent Performance Art of Alexander McQueen’s Collection No. 13—20 Years Later.” Vogue. 2018. https://www.vogue.com/article/alexander-mcqueen-no-13.

Mühe, Henrik, Andreas Angerer, Alwin Hoffmann, and Wolfgang Reif. 2010. “On Reverse-Engineering the KUKA Robot Language,” September.

Nagel, Christian. 2018. “Professional C# 7 and .NET Core 2.0 | Wiley.” Wiley.Com. 2018.

Naumann, M., K. Wegener, R. D. Schraft, and L. Lachello. 2006. “Robot Cell Integration by Means of Application-P’n’P.” In *In: Proceedings of ISR 2006*.

Nilsson, Klas, and Rolf Johansson. 1999. “Integrated Architecture for Industrial Robot Programming and Control.” *Robotics and Autonomous Systems* 29 (4): 205–26. https://doi.org/10.1016/S0921-8890(99)00056-1.

OCTOPUZ Inc. 2020. *OCTOPUZ*. https://octopuz.com/.

OpenMind. 2020. *HyperMill*. https://www.openmind-tech.com/.

OpenNURBS Initiative. 2020. https://www.rhino3d.com/opennurbs.

OpenTK. 2020. https://github.com/opentk/opentk.

Oxford Economics. 2019. “How Robots Change the World.” *Economic Outlook* 43 (3): 5–8. https://doi.org/10.1111/1468-0319.12431.

Pan, Zengxi, Joseph Polden, Nathan Larkin, Stephen Van Duin, and John Norrish. 2012. “Recent Progress on Programming Methods for Industrial Robots.” *Robotics and Computer-Integrated Manufacturing* 28 (2): 87–94. https://doi.org/10.1016/j.rcim.2011.08.004.

Parsons, John T., and Frank L. Stulen. 1958. Motor controlled apparatus for positioning machine tool. United States US2820187A, filed May 5, 1952, and issued January 14, 1958. https://patents.google.com/patent/US2820187A/en.

Pichler, Andreas, Markus Vincze, Henrik Andersen, Ole Madsen, and Kurt Häusler. 2002. “A Method for Automatic Spray Painting of Unknown Parts.” In *In IEEE Int’l. Conf. on Robotics and Automation*.

Pigram, Dave, and Wes McGee. 2011. “Formation Embedded Design.” In *Proceedings of the 31st Annual Conference of the Association for Computer Aided Design in Architecture*, 122–31. Banff.

PLCopen. 2018. “Creating Reusable, Hardware Independent Motion Control Applications via IEC 61131 3 and PLCopen Function Blocks.”

Quigley, Morgan, Brian Gerkey, Ken Conley, Josh Faust, Tully Foote, Jeremy Leibs, Eric Berger, Rob Wheeler, and Andrew Ng. 2009. “ROS: An Open-Source Robot Operating System.” In *Proc. of the IEEE Intl. Conf. on Robotics and Automation (ICRA) Workshop on Open Source Robotics*. Kobe, Japan.

Resnick, Mitchel, John Maloney, Andrés Monroy-Hernández, Natalie Rusk, Evelyn Eastmond, Karen Brennan, Amon Millner, et al. 2009. “Scratch: Programming for All.” *Communications of the ACM* 52 (11): 60–67. https://doi.org/10.1145/1592761.1592779.

RoboDK Inc. 2020. *RoboDK*. https://robodk.com/.

Rohmer, E, M Freese, and S. P. N. Singh. 2013. “CoppeliaSim (Formerly V-REP): A Versatile and Scalable Robot Simulation Framework.” In *Proc. of The International Conference on Intelligent Robots and Systems (IROS)*.

Rudolph, Jan-Peer, and Claus Emmelmann. 2017. “A Cloud-Based Platform for Automated Order Processing in Additive Manufacturing.” *Procedia CIRP* 63: 412–17. https://doi.org/10.1016/j.procir.2017.03.087.

Rutten, David. 2004. *VbScript|RhinoScript Workshop Handout*. McNeel.

———. 2020. *Grasshopper*.

Safeea, M., and P. Neto. 2019. “KUKA Sunrise Toolbox: Interfacing Collaborative Robots With MATLAB.” *IEEE Robotics Automation Magazine* 26 (1): 91–96. https://doi.org/10.1109/MRA.2018.2877776.

Schmidbauer, Christina, Titanilla Komenda, and Sebastian Schlund. 2020. “Teaching Cobots in Learning Factories – User and Usability-Driven Implications.” *Procedia Manufacturing* 45: 398–404. https://doi.org/10.1016/j.promfg.2020.04.043.

Schwartz, Thibault. 2013. “HAL.” In *Rob | Arch 2012*, edited by Sigrid Brell-Çokcan and Johannes Braumann, 92–101. Vienna: Springer. https://doi.org/10.1007/978-3-7091-1465-0_8.

Shah, Jami J. 2001. “Designing with Parametric CAD: Classification and Comparison of Construction Techniques.” In *Geometric Modelling*, edited by Fumihiko Kimura, 53–68. Boston, MA: Springer US. https://doi.org/10.1007/978-0-387-35490-3_4.

SharpDX. 2020. https://github.com/sharpdx/SharpDX.

SideFX. 2020. *Houdini*. https://www.sidefx.com/products/houdini/.

Siemens. 2020. *Parasolid*. https://www.plm.automation.siemens.com/global/en/products/plm-components/parasolid.html.

SPRUT Technology, Ltd. 2020. *SprutCAM*. https://sprutcam.com/.

Stanford, Nigel. 2017. *Automatica*. https://NigelStanford.com/Automatica/.

Steinhagen, Gregor, Johannes Braumann, Jan Brüninghaus, Matthias Neuhaus, Sigrid Brell-Cokcan, and Bernd Kuhlenkötter. 2016. “Path Planning for Robotic Artistic Stone Surface Production.” In *Robotic Fabrication in Architecture, Art and Design 2016*, edited by Dagmar Reinhardt, Rob Saunders, and Jane Burry, 122–35. Cham: Springer International Publishing. https://doi.org/10.1007/978-3-319-26378-6_9.

Steinhagen, Gregor, Jan Brüninghaus, and Bernd Kuhlenkötter. 2015. “Robotergestützte Künstlerische Steinbearbeitung.” In *Tagungsband Mechatronik 2015*, edited by T. Bertram, B. Corves, and K. Janschek, 103–8.

Stiehler, Alexander, and Sundeep Gantori. 2020. “Longer Term Investments: Automation and Robotics.” UBS. https://www.ubs.com/content/dam/WealthManagementAmericas/documents/automation-and-robotics-lti-report.pdf.

Sutherland, Ivan E. 1964. “Sketchpad a Man-Machine Graphical Communication System.” *Simulation* 2 (5): R-3.

Takeuchi, Yoshimi, Naoki Asakawa, and Dongfang Ge. 1993. “Automation of Polishing Work by an Industrial Robot: System of Polishing Robot.” *JSME International Journal. Ser. C, Dynamics, Control, Robotics, Design and Manufacturing* 36 (4): 556–61.

Thomson, Judith Jarvis. 1976. “Killing, Letting Die, and The Trolley Problem.” *The Monist* 59 (2): 204–17. https://doi.org/10.5840/monist197659224.

Thorsteinsson, G, and T Page. 2006. “Parametric Modelling in 3D CAD Systems Using Constraint and Variational Based Techniques.” *Society of Manufacturing Engineers, Dearborn, MI, USA, Technical Paper TB05PUB157*, 1–5.

Tolia, N., D.G. Andersen, and M. Satyanarayanan. 2006. “Quantifying Interactive User Experience on Thin Clients.” *Computer* 39 (3): 46–52. https://doi.org/10.1109/MC.2006.101.

Trower, Jake, and Jeff Gray. 2015. “Blockly Language Creation and Applications: Visual Programming for Media Computation and Bluetooth Robotics Control.” In *Proceedings of the 46th ACM Technical Symposium on Computer Science Education*, 5–5.

Vaisman, Lev, Laura Dipietro, and Hermano Igo Krebs. 2013. “A Comparative Analysis of Speed Profile Models for Wrist Pointing Movements.” *IEEE Transactions on Neural Systems and Rehabilitation Engineering: A Publication of the IEEE Engineering in Medicine and Biology Society* 21 (5): 756–66. https://doi.org/10.1109/TNSRE.2012.2231943.

Villemard. 1910. *En L’An 2000 - Chantier de Construction Electrique*.

vvvv group. 2020. *VVVV*. https://vvvv.org/.

Waibel, Markus, Bill Keays, and Federico Augugliaro. 2017. “Drone Shows: Creative Potential and Best Practices.” Application/pdf. ETH Zurich. https://doi.org/10.3929/ETHZ-A-010831954.

Waldt, Nils. 2005. “NC-Programmierung für das fünfachsige Flankenfräsen von Freiformflächen. – Institut für Fertigungstechnik und Werkzeugmaschinen.” Universität Hannover.

Wang, Huai, and S. Liu. 2014. “A Collision Detection Algorithm Using AABB and Octree Space Division.” *Advanced Materials Research* 989–994 (July): 2389–92. https://doi.org/10.4028/www.scientific.net/AMR.989-994.2389.

Weintrop, David, Afsoon Afzal, Jean Salac, Patrick Francis, Boyang Li, David Shepherd, and Diana Franklin. 2018. “Evaluating CoBlox: A Comparative Study of Robotics Programming Environments for Adult Novices.” In , 1–1. https://doi.org/10.1145/3170427.3186599.

World Economic Forum. 2016. “The Future of Jobs: Employment, Skills and Workforce Strategy for the Fourth Industrial Revolution.” http://www3.weforum.org/docs/WEF_FOJ_Executive_Summary_Jobs.pdf.

Zhao, Z., P. Zheng, S. Xu, and X. Wu. 2019. "Object Detection With Deep Learning: A Review." *IEEE Transactions on Neural Networks and Learning Systems* 30 (11): 3212–32. https://doi.org/10.1109/TNNLS.2018.2876865.

Zöller, Marc-André, and Marco F. Huber. 2021. "Benchmark and Survey of Automated Machine Learning Frameworks." *Journal of Artificial Intelligence Research* 70 (April): 409–72. https://doi.org/10.1613/jair.1.11854.

List of Figures